Faculty of Engineering University of Porto

Formal Advanced Mission Planning Specification

Pedro Filipe Lopes Maçaira Nogueira

Master Dissertation conducted within the Program of
Integrated Master in Electrical and Computers Engineering
Branch: Automation

DISSERTATION

Supervisor: Eng. João Tasso Borges de Sousa

July 31, 2013

U.PORTO
FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

MIEEC - MESTRADO INTEGRADO EM ENGENHARIA ELETROTÉCNICA E DE COMPUTADORES 2012/2013

A Dissertação intitulada

"Formal Advanced Mission Planning Specification"

foi aprovada em provas realizadas em 26-07-2013

o júri

Presidente Professor Doutor Paulo José Cerqueira Gomes da Costa
Professor Auxiliar do Departamento de Engenharia Eletrotécnica e de Computadores da Faculdade de Engenharia da Universidade do Porto

Professor Doutor Eduardo Resende Brandão Marques
Professor Auxiliar Convidado do Departamento de Informática da Faculdade de Ciências da Universidade de Lisboa

Professor Doutor João Tasso de Figueiredo Borges de Sousa
Assistente Convidado do Departamento de Engenharia Eletrotécnica e de Computadores da Faculdade de Engenharia da Universidade do Porto

O autor declara que a presente dissertação (ou relatório de projeto) é da sua exclusiva autoria e foi escrita sem qualquer apoio externo não explicitamente autorizado. Os resultados, ideias, parágrafos, ou outros extratos tomados de ou inspirados em trabalhos de outros autores, e demais referências bibliográficas usadas, são corretamente citados.

Autor - Pedro Filipe Lopes Maçaira Nogueira

Faculdade de Engenharia da Universidade do Porto

Resumo

Recentemente, os avanços tecnológicos têm levado ao aparecimento e proliferação dos sistemas multi-veículos. Os crescentes interesses de pesquisa nestes sistemas têm-se multiplicado em várias questões. O tema desta tese é o estudo associado com o desenvolvimento de um modelo para sistemas multi-veículos que será implementado como uma aplicação de planeamento, que irá abranger questões como "task allocation", "world modelling" e "networks topologies".

Em primeiro lugar, foi realizado um estudo inicial em sistemas multi-veículos com o intuito de compreender as principais motivações para a implementação desses sistemas e como são implementados. Simultaneamente, foi feito um estudo sobre o estado da arte, não só para estes sistemas, mas também para compreender os paradigmas do planeamento.

Em segundo lugar, o trabalho focou-se na compreensão de como cada agente do sistema deve ser caracterizado, de forma a fornecer informação essencial quando se define um estado do sistema. Este foi o ponto de partida para construir os modelos conceptuais iniciais de cada um. Depois de ter esses modelos especificados e tendo em conta o estudo efectuado sobre "networks topologies", foi desenvolvido um modelo para abstracção e mapeamento de operações de cooperação. Todos os modelos desenvolvidos foram fundidos, compondo assim um modelo final do sistema.

Depois de ter o modelo do sistema desenvolvido, este foi codificado utilizando uma plataforma "open source" para planeamento, programação e programação por restrições, chamado EUROPA. Esta plataforma permite construir planeadores e tem como objectivo facilitar o processo de integração de planeamento, programação e satisfação de restrições em aplicações para utilizadores finais.

Finalmente, o modelo foi aplicado em alguns cenários operacionais exemplo, a fim de provar a sua capacidade de planeamento de operações em sistemas multi-veículos e entender as suas vantagens e desvantagens.

Abstract

In recent years, the technological advances have led to the appearance and proliferation of multi-vehicles systems. The increasing research interests in these systems have multiplied onto different issues. The subject of this thesis is the study associated with the development of a multi-vehicle system model to be implemented as a planning application, which will cover issues as task allocation, world modelling and networks topologies.

At first, an initial study of the background on multi-vehicles network, in order to understand the main motivations on deploying multi-vehicles systems, and how these were deployed, was done. Alongside, a study on the State of the Art was done not only to these systems but also on planning paradigms.

Secondly, the work focused on understanding how each system agent should be characterized in order to provide essential information when defining a system state. This was the starting point to build the initial concept models for the agents. After having these models specified and regarding the study done on the networks topologies, a model to abstract and map cooperative operations was developed. All the developed models were merged into one compounding the final system model.

After having the desired system model, it was encoded using an open source platform for planning, scheduling and constraint programming, called EUROPA. It allows building planners and has the objective to facilitate the process of integrating advanced planning, scheduling and constraint reasoning into end-user applications.

Finally, the model was applied to some operational scenarios examples in order to prove its ability to output plans for multi-vehicle systems and understand its advantages and disadvantages.

Acknowledgments

I would like to thank my parents for whom I am deeply grateful for all the sacrifices made, providing me the opportunity to be where I am today, and for the robust education they gave me, which I am proud of and helped me made the person I am today. I also want to thank my brother for being an inspiration for me, through all the success he has achieved and the principles he transmitted to me as an older brother.

I thank my supervisor, Eng. João Tasso Borges de Sousa, for the opportunity, support and guidance and shared knowledge during the development of this work, as well as the LSTS people, who helped me through the technical difficulties, especially José Pinto and João Fortuna.

I want to give a special thanks to Rita for listening, encouraging and giving me motivation words when I most needed. She was very important in my success and in the future I hope to continue to achieve it and share it with her.

I would like to acknowledge my fellow students for all the friendship, stories and good disposition during these years.

Thanks to all my family, each one of you has inspired me in its own way.

For last, but not least, I want to thank my best friends for all the moments we have passed together. Hopefully we are young enough to write more histories together.

"The surest way not to fail is to determine to succeed."

Richard Brinsley Sheridan

Contents

List of Figures

List of Tables

Acronyms

UAV — Unmanned Aerial Vehicle

AUV — Autonomous Underwater Vehicle

ROV — Remotly Operated Vehicle

ASV — Autonomous Surface Vehicle

FEUP — Faculdade de Engenharia da Universidade do Porto (Faculty of Engineering University of Porto)

LSTS — Laboratório de Sistemas e Tecnologia Subaquática (Underwater Systems and Technology laboratory)

IMC — Inter Module Communication

XML — Extensible Markup Language

UAS — Unmanned Aerial System

Chapter 1

Introduction

Over the last decades, and due to robotics technology evolution, several unmanned vehicles have been developed for military, search and rescue operations. By definition a military operation is the coordinated military actions in response to a developing situation and these actions are designed as a military plan. This is, a formal plan for military armed forces, their military organizations and units to conduct operations, as drawn up by their commanders in order to achieve the objectives. These plans are generally produced in accordance with the military doctrine of the troops involved. It helps standardize operations, facilitating readiness by establishing common ways of accomplishing military tasks. Just as purely human operations need to be planned following standardized procedures, so combined operations between human, manned and unmanned vehicles need. It is known that a successful planning is a crucial step to achieve desired goals.

1.1 Objectives

This project will be developed with the help of "Laboratório de Sistemas e Tecnologia Subaquática" - Underwater Systems and Technology Laboratory (LSTS) as it falls within its development motivation: the development of tools and technologies for the deployment of networked vehicle systems.

There is an on-going trend towards the creation and development of multi-vehicle systems. These systems are characterized by being composed by heterogeneous groups of vehicles, manned and unmanned, sensors and operators. One of the biggest challenges concerning these systems is that the network, all the components described before, is a dynamic network and being able to understand the networks that are formed, i.e. the network topologies, and their behaviours is a great motivation.

This project will focus on developing a model specification to be implemented as an advanced mission planning system with the objectives:

- Standardize mission plans to facilitate the inter-operability between different unmanned vehicles as UAV's, AUV's, ROV's and ASV's in advanced cooperative operations

- Provide a new approach on defining a system state, at any time, and with it being able to understand the networks that are created

1.2 Thesis structure

The present document integrates all developed work, as well as the results and the conclusions obtained.

In Chapter 2, some technical and theoretical background is given about LSTS, its hardware, software tool chain, objective and achievements.

In Chapter 3Chapter 3, a bibliographic review and the State of the Art for multi-vehicle systems and planning paradigms is presented.

In Chapter 4, the problem is described and defined formally. The formalization of the problem is the starting point to the development of the work.

In Chapter 5, the problem is approached through a design work involving multiple iterations from an initial concept model to an actual model encoding.

In Chapter 6, the global results for the model application are discussed. The model is also evaluated and analyzed along with multi-vehicle important issues.

Finally in 0, the conclusion and discussion about further developments is presented.

Chapter 2

Background

This chapter provides background material on vehicles, on the PITVANT project and on the LSTS software tool chain.

2.1 Introduction

The Projecto de Investigação e Tecnologia em Veículos Aéreos Não-Tripulados (Research and Tecnology in UAV's Project) arises from a joint operation between Faculdade de Engenharia da Universidade do Porto (Faculty of Engineering University of Porto) and Academia da Força Aérea (Portuguese Air Force Academy), being the third stage of a greater project that began in 1996 in AFA. This stage began due to the achieved results in the early stages, started January 2009 and has a scheduled end to December 2015.

Some of the specific PITVANT objectives are to develop several technologies as:

- Cooperative control for multiple UAVs with mixed initiative
- Data fusion systems
- Navigation systems
- Vehicle-interoperability and the standardization of interactions

And to train personnel with the ability to define requirements, to operate and maintain UAV's [1].

The Laboratório de Sistemas e Tecnologia Subaquática (Underwater Systems and Technology Laboratory) have been developing, designing and building several autonomous and remotely operated vehicles as well as several tools with the goal of deploying networked vehicle systems for oceanographic and environmental applications [2][3]:

- Unmanned vehicles – UAV's, AUV's, ROV's, ASV's (section 2.2)
- Neptus (section 2.4.1)
- Dune (section 2.4.2)
- IMC (section 2.4.3)

Developing joint operations between LSTS, which focuses on the development of surface and underwater unmanned vehicles, and the PITVANT project specifies one of the more interesting operational scenarios (see Figure 1).

2.2 Unmanned vehicles

As stated earlier the LSTS has a wide variety of available unmanned vehicles developed entirely at FEUP, some in cooperation with AFA and other were only reworked from existing plantafroms [4]:

- AUV's — NAUV; LAUV SeaCon(Figure 2a), Xtreme(Figure 2b), Green, Black, Blue; Isurus (Figure 3);
- UAV's — UAV Alfa Series (01, 02, 03, 04, 05) (Figure 4a); UAV Pilatos 2; UAV Pilatos 3 (Figure 4b); UAV Lusitânia (Figure 4c);
- ASV's — Swordfish (Figure 5);
- ROV's — ROV-KOS (Figure 6a); ROV-IES(Figure 6b);

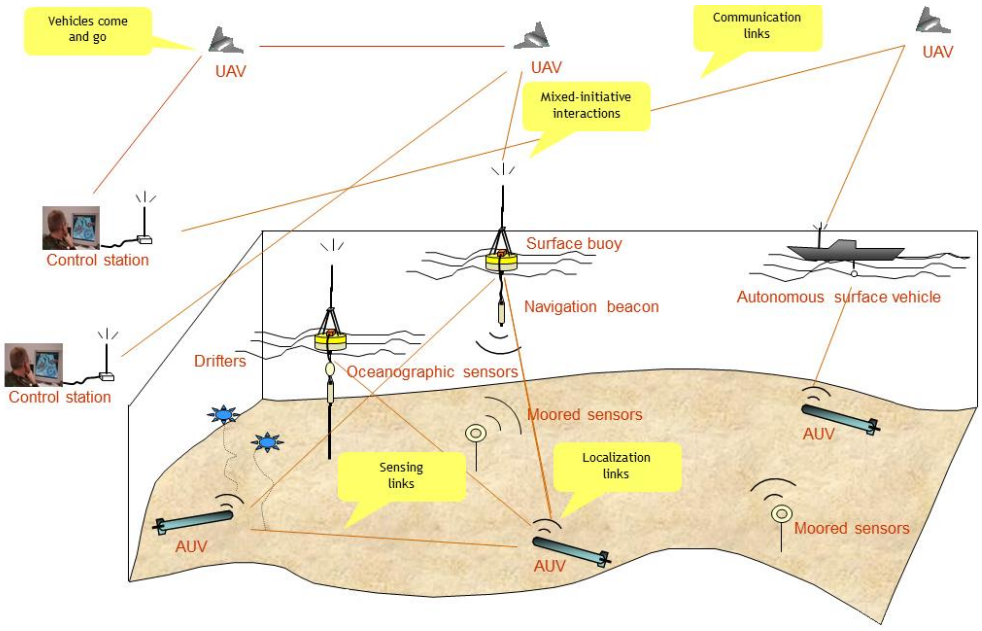

Figure 1 – Joint operation scenario between UAV's, AUV's and ASV's

a) LAUV Extreme b) LAUV SeaCon

Figure 2 – LAUV Models completely developed at LSTS-FEUP

Figure 3 – Isurus, a Woods Hole REMUS with reworked electronics and control software.

a) UAV Alfa 03 (Alfa series) b) UAV Pilatos 3 (COTS Model) c) UAV Lusitânia

Figure 4 – Examples of available UAV models at LSTS-FEUP

Figure 5 – ASV Swordfish, a commercial catamaran with added hardware and control software

a) ROV-KOS, completely developed at LSTS b) ROV-IES, reworked PHANTOM 500

Figure 6 - Examples of available ROV models at LSTS-FEUP

2.3 Control Architecture

LSTS has a layered approach to planning and executing control, and consists of two main layers: multi-vehicle control and vehicle control.

2.3.1 Single-Vehicle Control Architecture

Vehicles control architecture consists of four layers: low-level controllers, manoeuvre controller, vehicle supervisor and mission supervisor (Figure 7), and it is standard for all vehicles.

The concept of manoeuvre, an action/motion description for a single vehicle, is used as the atomic component of all execution concepts and it is abstracted from being provided by vehicles [5]. This concept plays a crucial role in the control architecture as it simplifies the task of mission specification; it is easily understood by a mission operator; it is easily mapped onto low-level controllers, since it encodes the control logic; and if defines clear interface to other control elements [6].

The vehicle supervisor controls all the on board activities and accepts manoeuvre and configuration commands either from the mission supervisor or external controllers. Interactions between external controllers and the vehicle are ruled by an abstract vehicle interface (section 2.4.1).

The mission supervisor, simply put, commands and controls the mission plan by exchanging manoeuvre and configuration commands to the vehicle supervisor that will trigger the execution of a manoeuvre by passing the manoeuvre parameters to the corresponding low-level controllers. Mission supervisor will only proceed with the next mission manoeuvre if it receives the completion acknowledgment of the previous one [5][7].

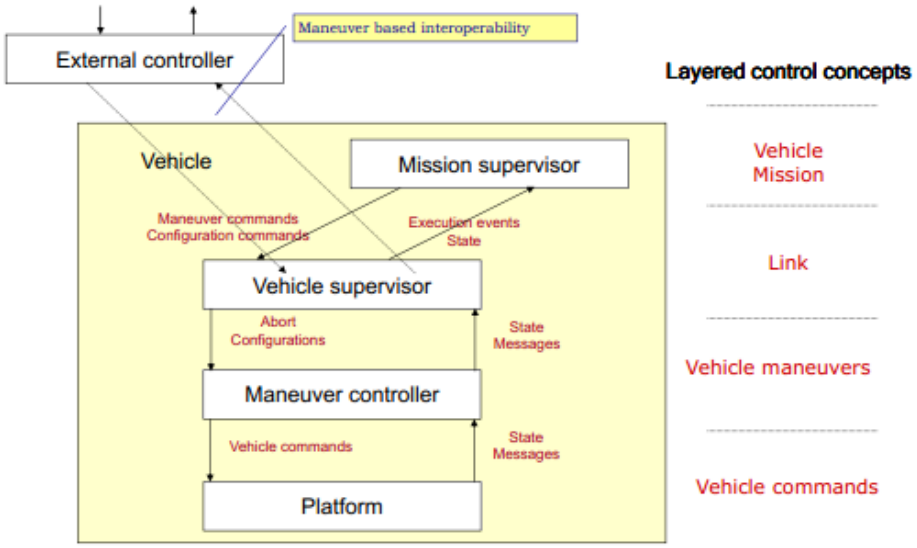

Figure 7 - Vehicle control architecture (adapted from [5])

2.3.2 Multi-Vehicle Control Architecture

The multi-vehicle control architecture is the extension of the single-vehicle control architecture with the introduction of another layer, the team controller layer, on the top on the previous ones. The team controller layer extends some of the concepts of the single-vehicle control architecture, sub-layers as the team supervisor and team manoeuvre that will command and supervise the execution of team manoeuvres. With this architecture the vehicle supervisor accepts manoeuvre and configuration commands from either external controller, plan supervisor or team controllers. [7].

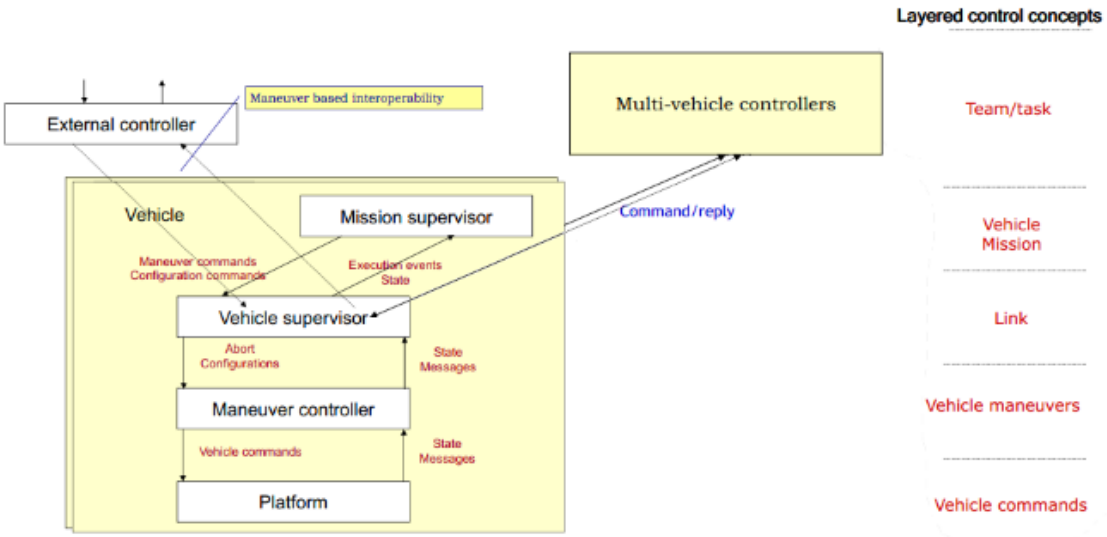

Figure 8 – LSTS layered control architecture (adapted from [5])

2.4 LSTS Software Tool Chain

2.4.1 Neptus Overview

Neptus is a Java based tool fully developed at LSTS-FEUP that allows the interaction with several manned and unmanned vehicles. It is a distributed command and control framework for operations with networked vehicles, sensors, and human operators and supports all phases of a mission life cycle: world representation; planning; simulation; execution and post-mission analysis [3].

This software has an Application Programming Interface (API) that provides different templates to create mission plans based on operators' inputs. These templates usually are available through "consoles" that implement the plugins. It is then possible to build new plugins that offer new functionalities and can be reused on different consoles allowing the creation of numerous consoles ideal for each type of mission or operator. This software's main communication interface is IMC (see section 2.4.3), what makes it interoperable with any other IMC-based peer, as all LSTS vehicles.

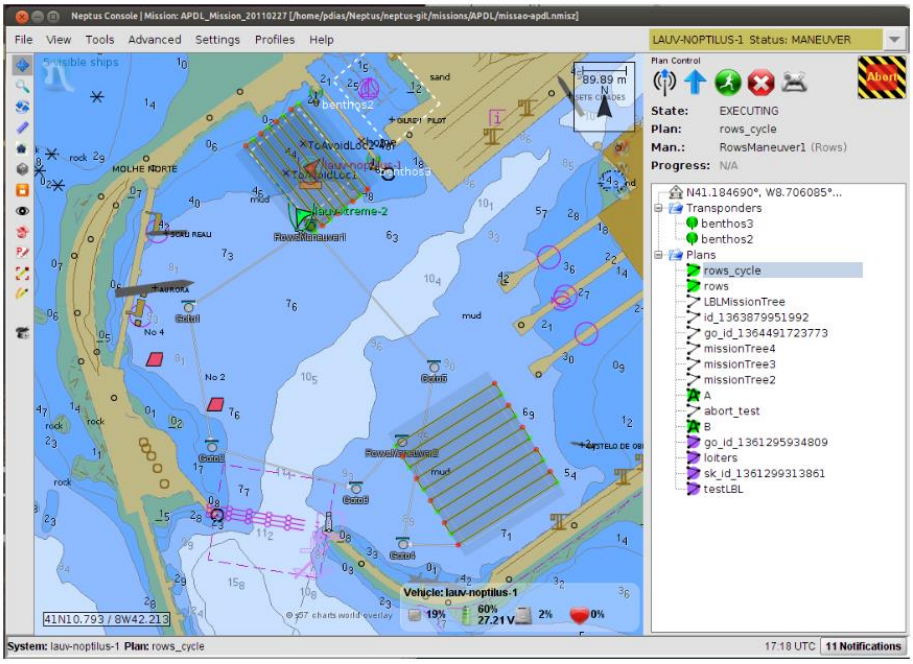

Figure 9 – Neptus operator console

2.4.2 Dune

Dune is a software tool developed at LSTS that integrates the UAS. This software runs in a small computer and has the ability not only to command the UAV but also has drivers for acquisition, navigation and manoeuvre control systems. Dune communicates with the ground station using the IMC protocol (see section 2.4.3).

2.4.3 IMC Overview

Inter-Module Communication (IMC) is a message-oriented protocol developed at LSTS-FEUP. It was developed for communication between different unmanned vehicles, sensors and mission management interfaces (external controllers) [8]. Figure 10 shows a typical message flow in an AUV.

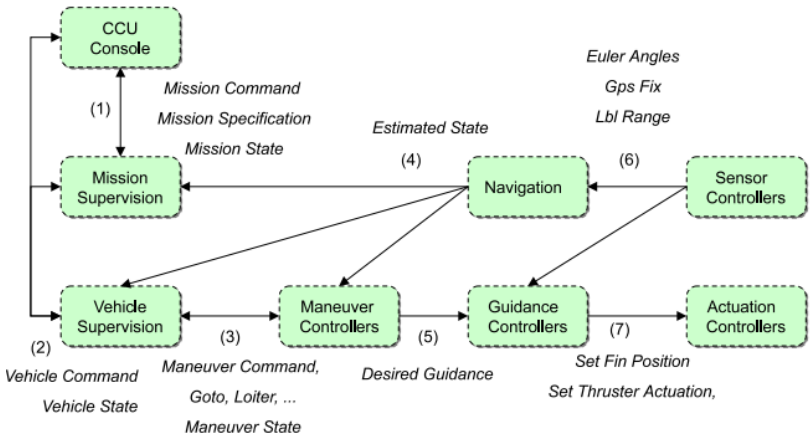

Figure 10 - IMC message flow in Seascout Light AUV [8]

Chapter 3

Literature and State of Art

In this chapter the concepts in which the developed system was built upon will be presented. Concepts brought up with multi-vehicles systems, some planning paradigms and a representation and problem solving framework are the main focus.

3.1 Multi-vehicle Networks

3.1.1 Introduction

In recent years, there have been increasing research interests in multi-vehicle systems. Several multi-vehicle research groups have diverged from this issue, being among them path planning [9–13], task allocation [14–19], world modeling [20] and networks topologies [21], [22]. Some of these groups' have been more excitingly explored and solutions for different kinds of problems have already been developed and optimized, as it can be seen in the paragraph bellow.

For path planning and trajectory generation, solutions have, for example, included the use of VORONOI diagrams [9], potential field theory [10], meta-heuristic search algorithms [11], grid model and bi-level programming [12], mixed-integer linear programming [13], among other. For task allocation problems, the so far developed solutions also include mixed-integer linear programming [16], [19] and meta-heuristic search algorithms [18] as well as auction algorithms [14], dynamic ranking [15] and central cognitive map approaches [17].

The world modeling and networks topologies research groups have recently attracted more interest from the researchers. Although not so much work can be found as for the groups presented above, interesting works can be found in [20] about world modeling and in [21], [22] about networks topologies.

This thesis will focus primarily on task allocation, world modeling and networks topologies problems. A more detailed and comparative evaluating approach will be done, in the next sections, to works of interest.

3.1.2 Task Allocation

Task allocation is the process that results in engaging specific tasks to specific vehicles, with the restriction that each task requirements should be satisfied, by the assigned vehicle, in order to be possible to achieve the task goal. In a simple mission configuration, with few vehicles, it is easy to a human planner to achieve optimal tasks assignments. However with the recent interest in deploying large multi-vehicle missions, this capability becomes too hard and time consuming. The transfer of responsibility from human planners to the vehicles themselves will improve the task allocation performance and reduce the manpower requirements [14].

As said in section 3.1.1, several methods, that empower vehicles with the ability to distribute tasks among themselves, have been developed. In [19] an early problem formulation using mixed-integer linear programming is described where a solution to a multi-task assignment problem is to be found. The attention-grabbing of this work is that the tasks are coupled by timing, precedence, and task order constraints. This allows variation of the vehicle path time to guarantee that the timing constraints are satisfied and directly incorporates the varying task completion times into the optimization. This formulation resulted in a large optimization problem with too many constraints to be applied to a real-time problem. At [16] the same problem formulation is used but with the inclusion of continuous timing variables that allowed solutions with any feasible task completion times to be calculated. The results presented were for practical problem sizes, but with larger problems, further work has to be done in order to simplify the problem structure and reduce complexity.

This reference [18] describes a kind of metaheuristic, the ant colony algorithm, that simulates the behavior of ants searching for food resources in nature. Optimizations to this algorithm, as the one that is presented in the work, are widely employed to solve optimization problems on task allocation. The model presented, as the ones presented above, also restricts the task allocation thought time, precedence and task order constraints. Additionally to this, the model also incorporates a performance requirement, given by the ratio of task covering, the cumulative distances travelled, the task completion time and the gaining maximal value. The optimization provided to the ant colony algorithm proved to find good solutions, especially with dynamic environments where tasks appear, largely due to the positive feedback the algorithm has.

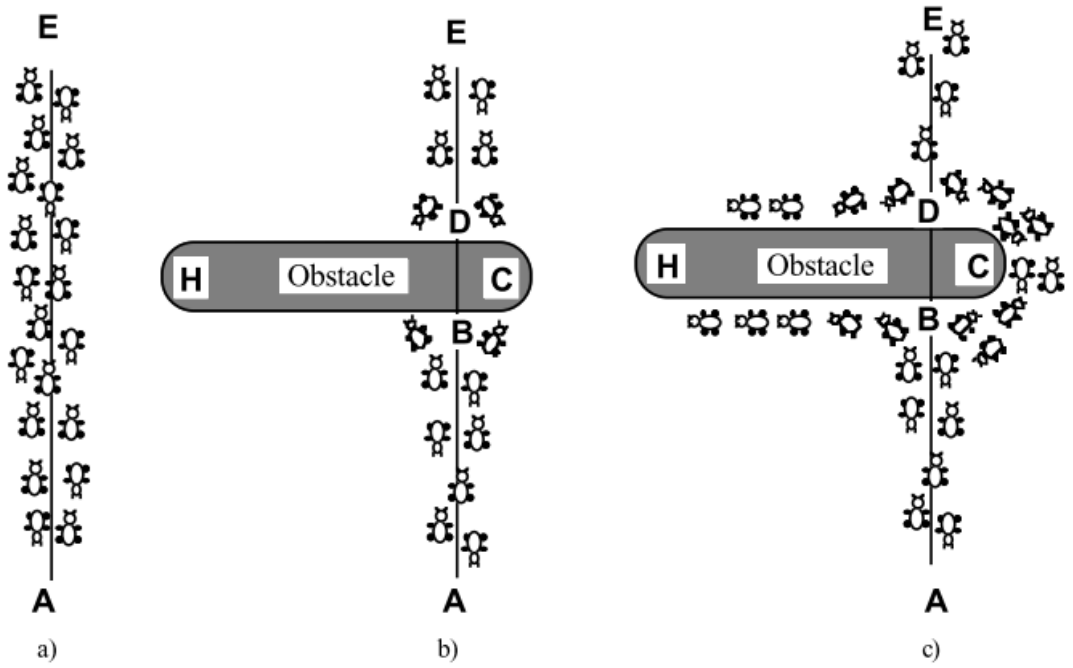

Figure 11 – Ant colony algorithm logic: a) Ants follow a path between points A and E. b) An obstacle is interposed; ants can choose to go around it following one of the two different paths with equal probability. c) On the shorter path more pheromone is laid down. [23]

Until here, the approaches described have one common feature: a centralized cooperative task allocation. In [15] a fully decentralized cooperative task allocation approach is proposed and it is based on a dynamic task ranking (DTR) procedure. It has two key concepts: agent-task benefit and task-task benefit. Each agent is assumed to be able to calculate the cost it takes to accomplish a certain task, so that if it is not capable, the cost is infinite. The benefit agent-task is the multiplicative inverse of the task cost, and the task-task benefit is the benefit that an agent has to accomplish the t^{th} task after the completion of the j^{th} task, defined in analogy with the agent-task benefit. So, in simple words, the benefit of accomplishing a task depends not only on its accomplishment, but also on the already accomplished tasks. For example, if there are nearby tasks, and an agent accomplishes one of those, it will be favored to accomplish the others. The approach presented does not define formally the tasks, it is only known that there are tasks to be executed and that each agent is able to calculate the costs.

Another decentralized task allocation method is presented at [17]. A simple model where vehicles choose tasks autonomously in real-time, using a central cognitive map, is described. This model was used considering a heterogeneous group of vehicles drawn from distinct classes. The idea behind the central cognitive map is to provide instantaneous and accurate information of the current mission situation to the vehicles team. The model defines the vehicles state, tasks state and the task assignment state through state vectors. A $L_x \times L_y$ cellular environment is considered, where each cell state is given by the target occupancy probability (TOP) value, which defines the estimated probability that cell contains a target. At every time, t, every cell (x,y) in the environment has an associated task, and each cell

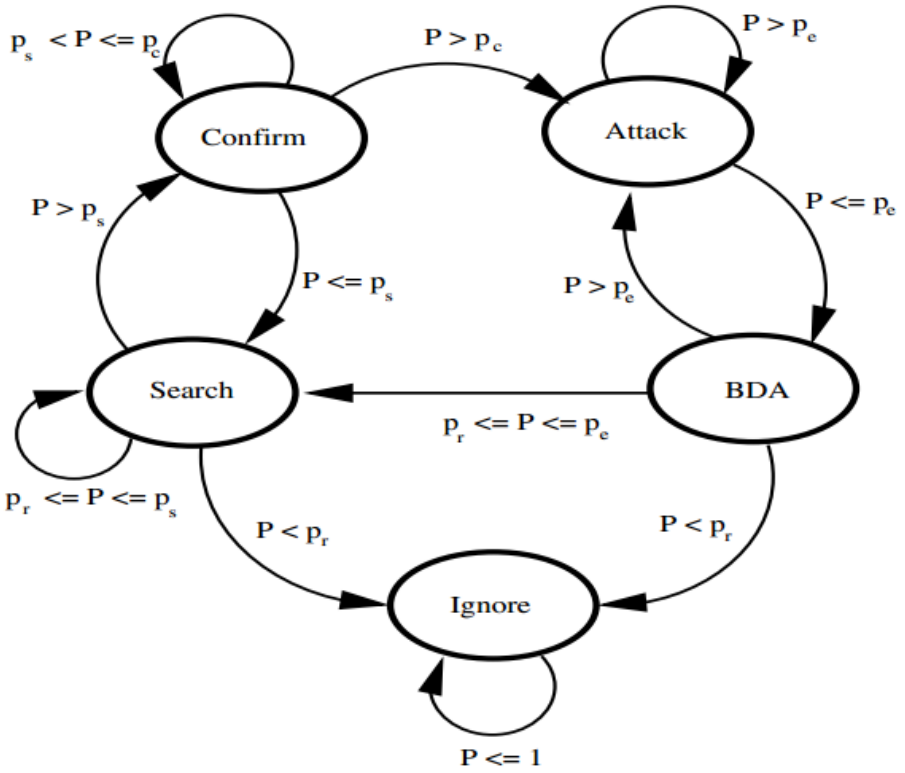

Figure 12 - Task Dynamics. ps=suspicion threshold; pc=certainty threshold; pe=exit threshold; pr=resolution threshold [17]

state defines the task action that needs to be done. Every time a vehicle performs a task action in a cell, the TOP value is updated through defined probabilistic functions, and according to the task dynamics thresholds (Figure 12) the cell state will change. At each step the vehicles report their actions to a centralized information base (IB) and update their knowledge off the environment by reading the IB.

Interesting contributions are found in this work, more precisely, the fact that vehicles states, tasks states and tasks assignment states are defined by state vectors, which proves to be an easy approach on sharing states information among information structures. Although, a limitation to this approach could be pointed out: all vehicles have to continuously maintain communication links to the IB in order to update and read it.

3.1.3 World Modeling

Efficient operational autonomous systems require a comprehensive overview on their environment [24]. This statement portraits the importance of the world modeling issue. With the recent grow of multi-vehicle systems it becomes clear that having a formal representation and understanding of the relevant surrounding world is a key element for achieving successful missions. Obviously, this representation has to be common to all system agents and they should be able to maintain and share a real-time world model.

Reference [20] describes a world modeling approach for cooperative intelligent vehicles. The proposed approach uses Local Dynamic Map (LDM) as the element responsible for representing and maintaining a real-time world model. LDM is an object oriented representation that contains both real life and conceptual objects, being all characterized by attributes, uncertainties and some relevant relationships between them (see Figure 13). The model is shared by all onboard control functionalities, denominated as "Applications", and works as an abstract layer between them and the data interpretation (see Figure 14).

The proposed architecture is described as having a number of advantageous features, being some of them, in this work point of view, considered important contributions:

- Sensory and control aspects are isolated by an abstract layer
- The vehicles can, through dedicated applications, share local world models among them

Another contribution, which should be pointed out, is, as said earlier, that everything considered important from the real world is modeled as an object. Object types are defined in an inheritance hierarchy which not only allows an easier categorization management, as well as an easier extendibility, without disturbing the existing ones.

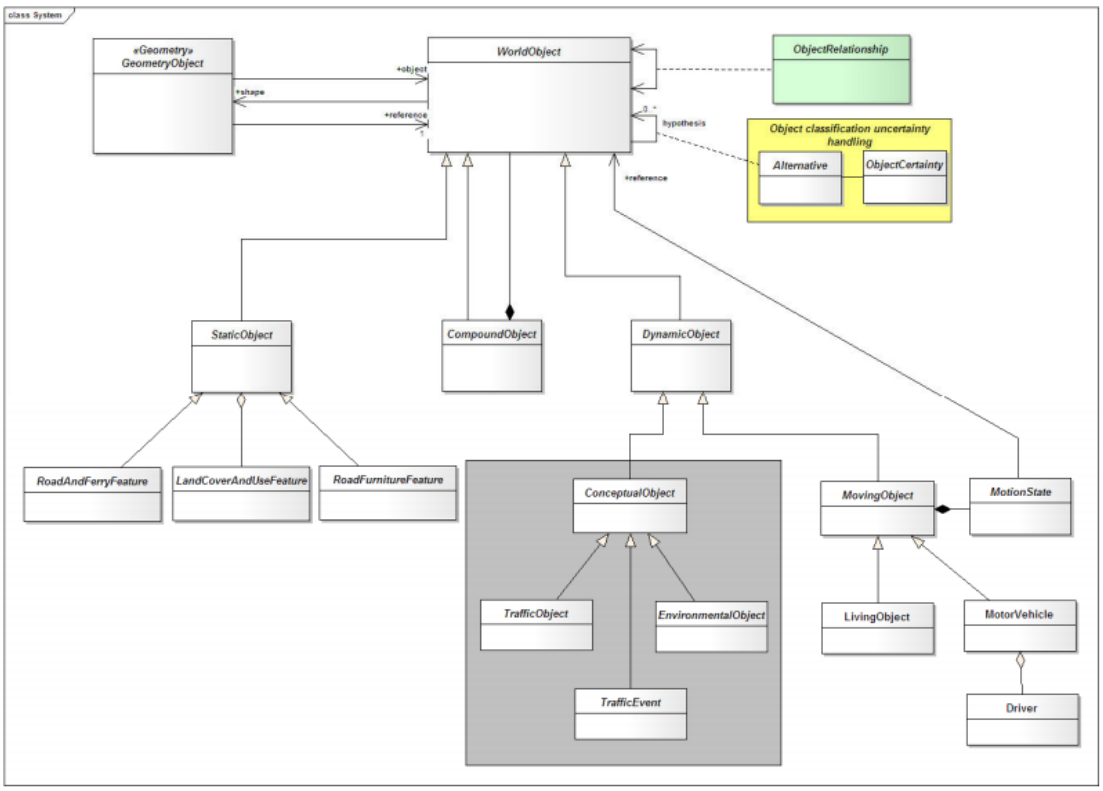

Figure 13- LDM object model

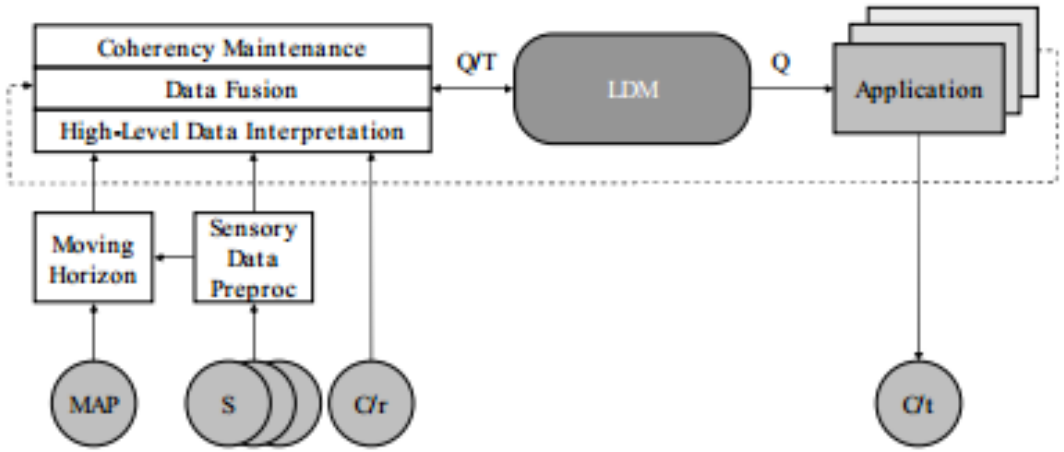

Figure 14 – LDM proposed architecture

3.1.4 Networks

In multi-vehicle systems, a network topology should describe, logically or physically, the arrangement of the network, by describing its nodes and connecting lines. As these systems are growing and becoming large scale systems, it becomes imperative to develop new methods that will provide additional support on vehicles' cooperation and coordination.

In [22] a service network model is described with the objective to find an organizational paradigm that will fit to multi-vehicle and multi-task systems. The model consists of two basic entities: services and service providers. Service providers deploy and make theirs services available to others, being, typically, the deployment of a service dependent on other providers' services. So, it can be claimed that this service network consists of a set of service providers that use each other services. Service providers can be physical entities as control stations, vehicles, among other or logical resources as complex algorithms. This model was applied to a simple multi-vehicle search mission, where a mission control requests a search service, with coordinates and object description as service constraints. The search service provider then divides the territory and requests sweep services to multiple vehicles. There is a reliable data storage provider to share information among all vehicles.

Provider	Service	Service Request
p1	search	{{import(p1,sweep,__,{},{}}, {}}
h1	sweep	{{},{}}
h2	sweep	({},{})

Figure 15 – Example of simple knowledge base [22]

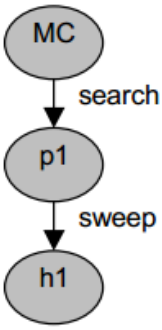

Figure 16 – solution generated by the algorithm [22]

A service network can be represented as a directed rooted graph. The addressed problem consists on finding a graph G, which includes the initial graph G0, being consistent, complete, and connected, as well as the minimal solution to the problem. The initial graph G0 represents the service requirement and associated with it has the requester constraints. Each service provider has some constraints associated with each service it deploys.

The main contributions to learn and keep from this work are: network topologies can simplify the achievement of cooperative multi-vehicle configurations, distributing objectives and at the same time sharing knowledge; networks topologies defined through abstract capabilities, as the sweep and the search service, can be applied to various types of vehicles; and although the model is not referred to as a task allocation model, it in fact distributes tasks according to the requester restrictions, maybe not achieving the best solution, in terms of for example cost, but satisfying the requester objectives.

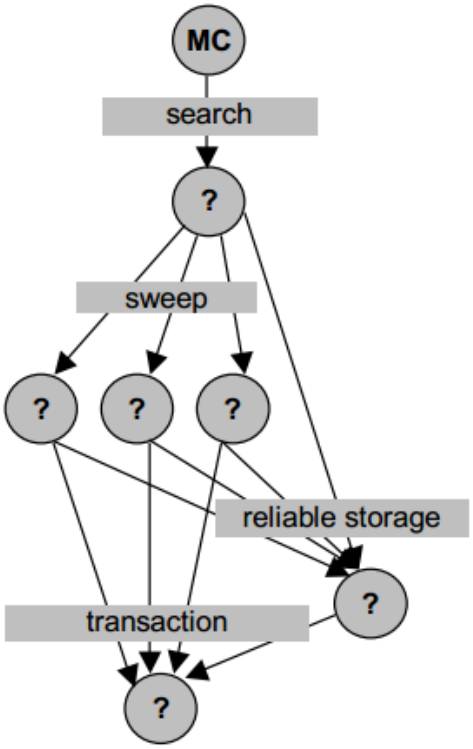

Figure 17 – SN developed to the multi-vehicle search; the "?" correspond to the unknown service providers that the solve primitive is supposed to fill in [22]

3.1.5 Multi-vehicle Systems Issues

Multi-vehicle systems, at some point, become coupled distributed networks of semi-autonomous problem solving agents. For example, if a problem can be solved more effectively through cooperation, the vehicles should work together by dividing the problem into sub problems each should solve. Depending on the problem, sub problems can be dependent and sequential, and, in that situation, agents must coordinate their local problem solving actions. The main problem solution will, in any circumstance, arise from the combination of the sub problems results [25].

What will be presented in this section are the main issues concerning this type of multi-vehicle cooperating systems. Four main issues were identified: System model; System domain an environment assumptions; Fault tolerance; and Communication importance. A more detailed work on evaluating and comparing cooperative distributed problem solving researches is done in [25]. The issues presented will work as guidelines when evaluating and classifying the work developed (0, section 7.2):

- System model: this issue focuses on describing and representing the system. The following questions will help understanding each problem solving model:
    - Is there a defined control hierarchy? How is it defined?
    - Does cooperation exists? Is it crucial in achieving system goals?
    - Does cooperation between agents forms an agent composed model? What benefit comes from achieving the composed model? Who knows when a model is composed?
    - What and where are the communicating structures? What connections are needed? Do they need to be permanent?
    - Problem-solving Strategy: How are system goals distributed and broken down to sub goals? Are all goals defined in the planning phase?
- System domain and environment assumptions: this issue focuses on describing the assumptions that the problem solving model makes about the problem domain. It is important to state that the less assumptions, the more generally applicable the system is to problem solving [25].
    - How is the system domain characterized?
    - What a priori knowledge do agents have about other agents?
    - How is problem solving knowledge shared among agents?
    - Is the system domain dynamic? What features may vary along time?
- Fault tolerance: this issue focuses on describing how the system deals with multiple kinds of failures and if it includes mechanisms for problem solving in real-time domains.
    - What possible failures are assumed to be possible?
    - How does the system deals with faults?

  - Is predictive timing information available?
  - Are there tasks deadlines?
- Communication importance: this issue focuses on describing how communications are made in the system and how agents know when to communicate and to whom.
  - How are the communication paths specified?
  - How are messages addressed?
  - What information do agents use to determine the when, whom, and what of communication?

3.2 Planning Paradigms

3.2.1 Planning, Scheduling and Automated Planning

Traditional planning often views planning systems as simple isolated components that accept a set of goals, initial conditions, and a description of the possible actions that can be executed and generate an output, as seen in Figure 18. This output is called a plan, a sequence of actions, that after executed makes the system to achieve the goals [26]. This idea gives the simple definition of planning: finding a sequence of actions that achieves a desired goal.

Scheduling can be thought of as determining whether the needed resources are available to complete the plan. It concerns with the allocation of resources to activities with the objective of optimizing some performance measures. Resources, depending on the situation, can be viewed as a variety of things as machines, humans, fuel, battery, etc. and the scheduling objectives could be minimization of the plan length, maximization of resource utilization, among other [27].

Automated planning is the area of artificial intelligence (AI) that studies the deliberation process described above computationally. In automated planning, the problems states correspond to instantaneous descriptions of the world and the actions that an agent can perform to change the state of the world [28]. The state is usually represented as a set of

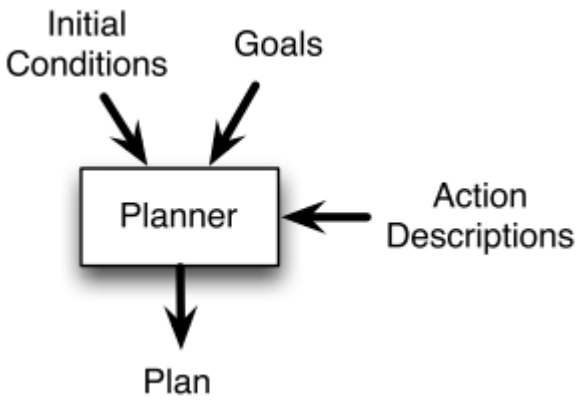

Figure 18 - The traditional view of planning as an independent component [26].

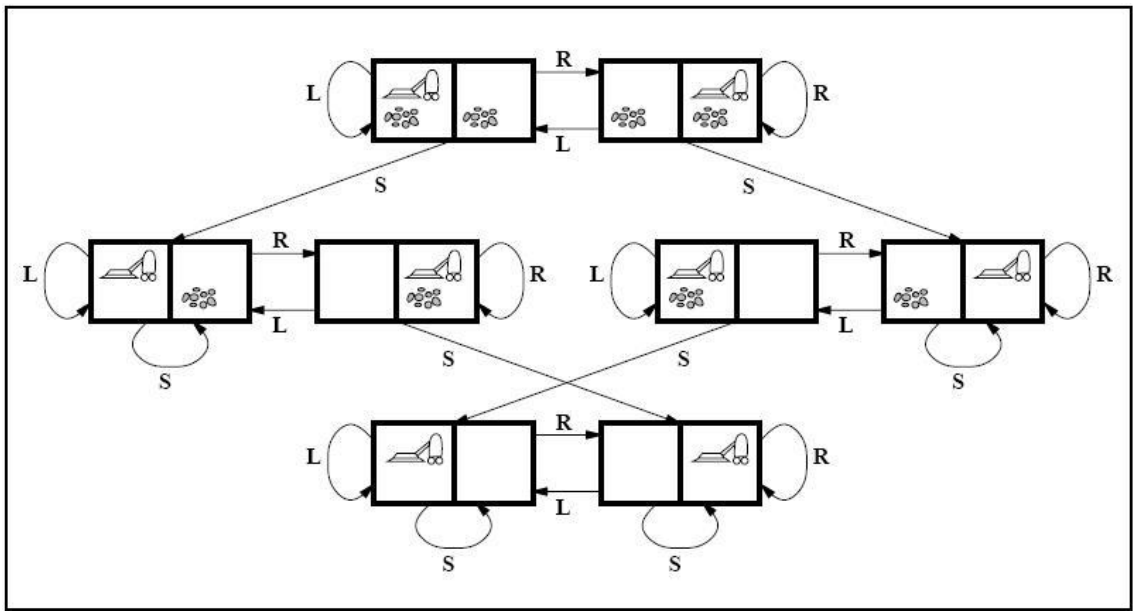

Figure 19 – The state space for the vacuum world. Links denote actions: L=Left, R=Right, S=Suck.

logical formulas and the state transitions implies the change in formulas. How the world changes through actions is described in the domain model. In Figure 19 it is possible to see the state space for a vacuum world toy example where:

- State – determined by the agent location and if the locations are clean/dirty, being 8 possible states. The agent is in one of two locations, each of which might be clean or dirt
- Actions – Move Right, Move Left, Suck
- Transition Model: actions have their expected effects, except that moving Left in the leftmost square, moving Right in the rightmost square, and Sucking in a clean square have no effect.
- Goal – Clean everything

3.2.2 Constraint-Based Planning

Constraint programming was born as a multi-disciplinary research area that uses techniques and notions from many other areas as artificial intelligence, computer science, databases, and programming languages, among other. It is applied with success to variety of domains, including scheduling, planning, networks and vehicle routing [27].

Constraint programming is a programming paradigm where the relations between variables are stated in the form of constraints and a constraint satisfaction problem states which relations should hold among the given decision variables [27]. A simple example is provided next, adapted from [29]. Having the following variables:

- Speed = [1 10] i.e. the variable speed has a value in the range from 1 to 10.
- Distance = [40 100] i.e. the variable distance has a value in the range from 1 40 to 100.
- Time = [0 inf] i.e. the variable time has a value in the range from 0 to infinity.
- Location1 = [20 25] i.e. the variable location1 has a value in the range 20 to 25.
- Location2 = [80 200] i.e. the variable location2 has a value in the range 80 200.

And the set of constraints:

- C0: speed == distance/ time
- C1: location1 + distance == location2

The solution for this problem is given each variable a value such that all constraints are satisfied. A possible solution would be:

- speed=10; distance=70; time=700; location1=25; location2=95.

3.2.3 Simple Temporal Problems (Networks)

A simple temporal problem is a problem where all the constraints specify a single interval. In [30] it is proposed that constraints among time points can be grouped together to form a Single Temporal Network (STN). This network can then be transformed into a Distance Graph (DG) where the external arc from a node to a target node represents the maximum distance. In Figure 20 it is possible to see a STN with 2 variables and a single constraint, and the resulting DG (adapted from [29]).

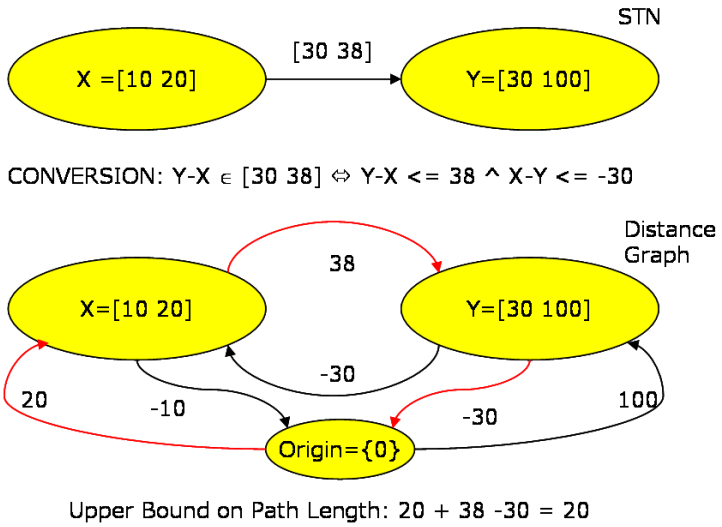

Figure 20 - STN with two variables and a single constraint; Resulting DG

3.2.4 Constraint-based Attribute and Interval Planning

Constraint-based Attribute and Interval Planning (CAIP) was described in [31] as a planning framework that supports features common to real planning problems. It provides primitives that support modelling domains with real time, concurrency, resources, mutual exclusion, and disjunctions. This framework has a representation of temporally extended states, named intervals, which provide a basis for constraining timing and concurrency of activities and, the notion of attributes to enforce mutual exclusion and to support resources modelling. Its fundamentals are in constraint-based planning (section 3.2.2) which permits compact representation of these rules, supports disjunctions, and also planning technology to leverage off efficient algorithms for constraint satisfaction problems.

This framework has already been implemented in different systems, being one of them the EUROPA system that extends the application of this technology to real-world planning domains as well as the capabilities of the CAIP paradigm [31]. The EUROPA system will be depicted in the following section.

3.3 Europa

3.3.1 Introduction

The origin of EUROPA, an open source platform for planning, scheduling and constraint programming, is derived from the HSTS planner, a representation and problem solving framework that aims at unifying planning and scheduling [32]. It is a class library and tool set for building planners within a constraint-based temporal planning paradigm (section 3.2.4). Its objective is to facilitate the process of integrating advanced planning, scheduling and constraint reasoning into end-user applications [33]. As seen in [34–37] it has been applied to a wide variety of practical planning problems and it is clear that EUROPA was designed to support planning for complex systems, among them unmanned vehicles.

3.3.2 Technical Background

EUROPA, as is the case with is predecessors, uses a domain model (Figure 21) written in a declarative language together with initial conditions and goals in order to construct a set of temporal relations that are required to be true at start time [38]. This means that a single planner can be applied to different systems and problems if different models and goals are provided. These declarative planners rely on a very expressive language for models, goals and plans [29].

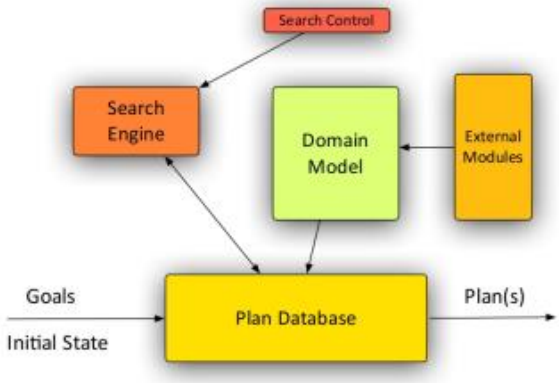

Figure 21 - A general architectural block diagram for an AI based Planner [38]

In the traditional approaches for managing complex systems the planning and scheduling phases are very distinct [32], [33], [38], as also said in section 3.2.1.

The first is considered to be the process of generating descriptions of how the systems achieve desired goals. These descriptions consist of connecting elementary actions to move the system into a state that satisfies the goal and are called plans. In order to be possible to generate plans for a desired system a model of how the system works must be given.

The other phase, scheduling, results in a prediction of a specific sequence of actions that ensures the achievement of all goals within the system's physical constraints. A scheduler instantiates the plans and assigns to each action a time slot for the exclusive use of the needed resources.

In Figure 22 it is possible to see a traditional planning approach to a simple domain problem. The traditional planning describes models in terms of a set of Boolean state indicators ("fluents") and a set of actions that can modify them.

This approach is too simplistic for modelling real-world applications as unmanned vehicle systems that consist of multiple sub-systems interacting, that may be active at the same time and performing different tasks. To plan for these systems the expressiveness of the model, plan and goals descriptions must be significantly improved [29], as in EUROPA descriptive language. This leads to some divergences between classical planning and EUROPA.

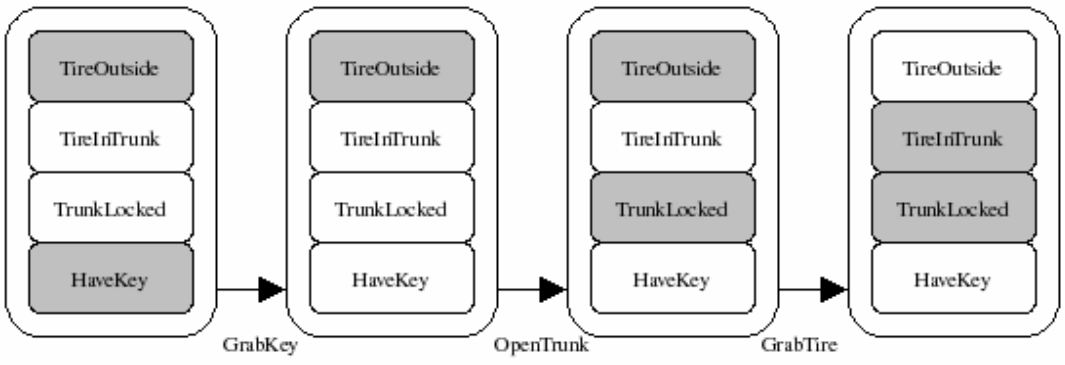

Figure 22 - The Tire-World Domain - an example of a sequential operator plan. States contain fluents that are true (white) or false (greyed-out). Actions effect plan state. [29]

One divergence is that EUROPA uses a state variable representation [29], [32] to describe the evolution of state overtime using the same predicate logic formalism to describe states and actions having no distinction between the two. These state variables are called timelines and the values of timelines are sequence of procedures [33] that encapsulate and describe state evolution. Predicates are used to describe things that are true, having no description for false states. Consequently, EUROPA relies on scoping the temporal extent over which a predicate holds, being an instance of a temporally scoped predicate called a Token.

Since EUROPA is based on a constraint-programming model (section 3.2.2) the relationships between tokens are expresses as simple arithmetic constraints over the temporal variables, following Allen's relations [39]. The possible relations are shown in Figure **23** and in Figure 24 it is possible to see a simple timeline where tire is an object in the tire domain world and it has predicates associated with it which describe its states and actions

.

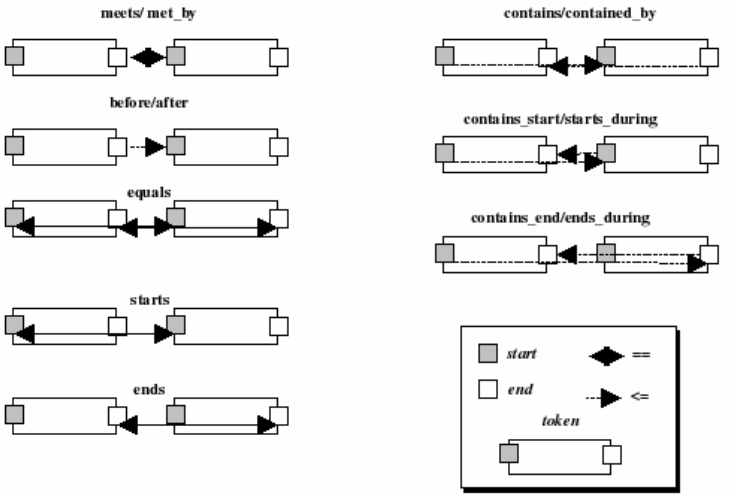

Figure 23 – **Possible token temporal relations** [29]

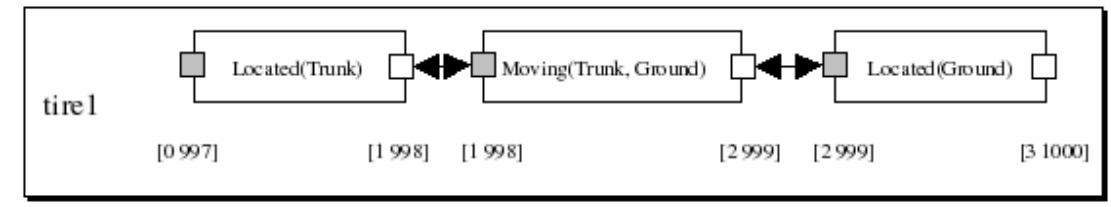

Figure 24 - A Timeline for a tire. Located is a state and moving an action [29]

3.3.3 Plan Representation

EUROPA follows the representation outlined by CAIP (see section 3.2.4) with elements that are required for an efficient implementation of constraint-based planning [38]. Here the main elements, which are built upon the framework of Constraint Satisfaction Problems (see

section 3.2.2) and the related work of Simple Temporal Problems (see section 3.2.3), are introduced [29], [38]:

- Domains, Variables and Constraints – used to describe the problem in terms of a CSP problem
- Tokens – an instance of a predicate with defined temporal extent used to represent actions and states
- Objects – things to be described and referred to in a domain
- Timelines – provides simple method for aggregating the statements about an object
- Token State Model – supports an efficient implementation of plan-space search algorithms
- Rules – for a plan to be valid, it must comply with all the rules pertinent to the relevant application model

3.3.3.1 Domains, Variables and Constraints

Variables and constraints are the basic building blocks of EUROPA. Variables are values that need to be represented to describe the problem domain over which constraints may be specified. They can be introduced in different ways and in a range of scopes [29], [38]. A variable can take the value of a domain, specific value (singleton) or interval (only for numeric data types). Domains can be defined over the following data types: String, Boolean, Numeric and Object. Constraints are used to represent the restrictions over which a plan must be validated and can be defined over any variable combination in a domain.

3.3.3.2 Tokens

Tokens are instances of a predicate with defined temporal extent used to represent actions and states. In general, for an execute plan it is not enough to define predicates without giving them some temporal extent over which they hold. If a predicate is always true it can be thought to hold from the beginning to the end of the planning horizon although in practice the temporal extend of interest must be defined with starting and ending time points. Looking at the example shown in Figure 24, it could be written “Located(Trunk,0,998)” to specify that the tire is located at the trunk from time 0 to time 998 but as this pattern of using such predicates to describe both state and behaviour of

objects is so predominant in EUROPA, tokens have been introduced as special constructors with built-in variables [29], [38]:

- Start – the beginning of the temporal extent over the predicate is defined
- End – the end of the temporal extent over which the predicate is defined
- Duration – the constraints start + duration = end is enforced automatically
- Object – the set of object to which a token might apply
- State – tokens can be ACTIVE, INACTIVE, MERGED, or REJECTED, capturing the token's current state and its reachable states through further restrictions.

3.3.3.3 Objects

Objects are the things to be described and referred to in a domain. As in object-oriented modelling, objects can be found by seeking out the nouns in any domain description. An object is an instance of a class and it is modelled using the abstraction of a class to express the fact that all instances have certain properties of state and behaviour. These are built on the formalism of first order logic [29], [38]. As can be seen in Figure 24, where an object tire from the tire domain world can be Located on a position or Moving from one location to another.

3.3.3.4 Timeline

A timeline is a EUROPA built-in class from which objects can be derived from and inherit special features. Objects derived from timelines will induce ordering requirements among its tokens to ensure no temporal overlap may occur among them [29], [32], [38]. Again looking at Figure 24 it is possible to verify that a token cannot be located at both places at once.

3.3.4 Modelling

As said in 3.3.2, EUROPA can be applied to different domains and problems if different models and goals are provided having consequently to rely on a very expressive language for modelling. EUROPA's main input modelling language is the New Domain Definition Language (NDDL), a domain description language for constraint-based planning and scheduling problems [33]. NDDL allows users to specify the domain world elements in a precise and concise way, being the main feature of it the fact that it is object oriented [38]. A complete NDDL reference guide with examples and the NDDL grammar guide is available at [29].

Chapter 4

The Problem

With the advanced developments in unmanned vehicles, large and heterogeneous systems are being deployed. These systems consist of many resources interacting to successfully execute operations and achieve operations goals. It is known that heterogeneous systems can offer more features and capacities than homogenous systems itself, largely due to the several composed operations between system entities that can happen. As these are highly dynamic systems, at any time, the features they offer can vary.

The big issue born with these large scale heterogeneous systems is how to capture the system state, at any time, given that there are several entities, interacting and performing operations at different levels. It is important to mission supervisors and planners to know what is happening in the world and what are they able to do or request, given that the system offers a set of features for limited time.

Another issue is how to define the composed operations that can happen in these systems. This issue comes along with the stated before because, on a high-level supervising perspective, it is not crucial to know what a single actor is providing to the system but what a composed operation is. Given this, it becomes critical to define and specify these composed operations and also provide the systems with the ability to identify what they can offer by coordinating and combining the existing available resources, and therefor achieve known composed operations.

4.1 Statement

What is to be developed in this work is a new model for defining composed operations and a new system state approach. The objective of this model is, not only to help defining a system state, at any time, but also to provide specific entities, of these heterogeneous multi-tasking systems, with the planning knowledge to take advantage from having the right resources to establish composed operations.

4.2 System Specification

4.2.1 System State

In order to solve the system state specification issue, firstly it is important to identify the key elements to define a system state at any time. A system state should be described by all the system components, either being:

- Physical entities (PE)
- Computational Entities (CE)
- Abstracts Entities (AE)

and each of these elements, should have a state representation that combined together with all the existing elements state representation, define the system state. Consequently, along with identifying the key elements in a system state, it is imperative to define how to characterize each one, in order to represent its state over time.

Physical entities by definition are entities with physical existence. In heterogeneous multi-tasking systems these entities are mainly control stations, vehicles and their payload. As payload is associated with bigger entities, as vehicles, its characterization becomes part of the main entity characterization.

Computational entities, as the name suggests, are computer programs that act for a user or other program in order to solve problems. In the studied systems, these are controllers and planners, which run on physical entities and therefore will be included in their characterization.

Abstract entities are defined by not existing as physical or computational entities, but rather as an idea, type of thing, that have been condensed from concrete realities. In this work, interactions between physical entities, computational agents and the environment can be abstracted as being from a specific type. These abstract entities have to be associated with physical entities, in order to provide them with the knowledge of what they are doing in a high-level mission perspective. In other words, in these heterogeneous, large-scale, multi-tasking systems, it becomes significantly important to know not only what a single actor is providing, but what a cooperative group of actors is providing, by being in some composed operation over time.

At this point it comes clear that having a well-defined characterization for physical entities and thereby their components, either being computational agents or other physical entities, is crucial for having a well-defined system state at any time. Figure 25 illustrates the characterization architecture for main physical entities.

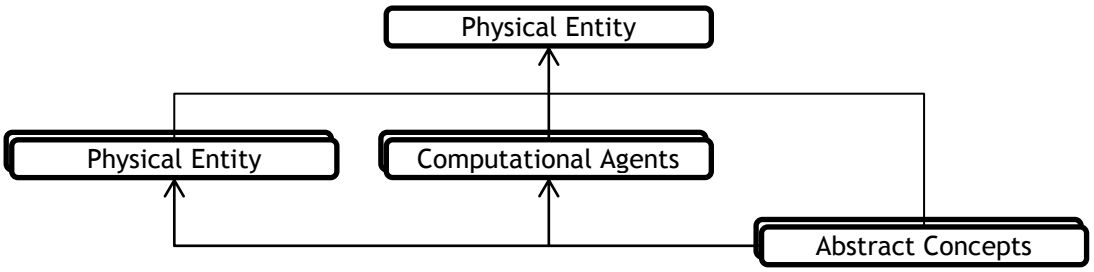

Figure 25 – Physical entity characterization architecture

It can be claimed that gathering and merging the states of all existing physical entities on a system, at any given time, provides a consistent system state for that time.

4.2.1.1 Physical Entities Characterization Requirements

Each of the main physical entities that have been identified, vehicles and control stations, should be defined by a specific characterization that focuses on describing the entity main properties and its state properties in the world.

The main properties, although not being related to the entity state, because they should be unchanged over time, are an important component of the entity characterization, because they should express the entity configuration.

The state properties considered to be crucial, when capturing the entity state, should be identified based on the needs of experienced operators.

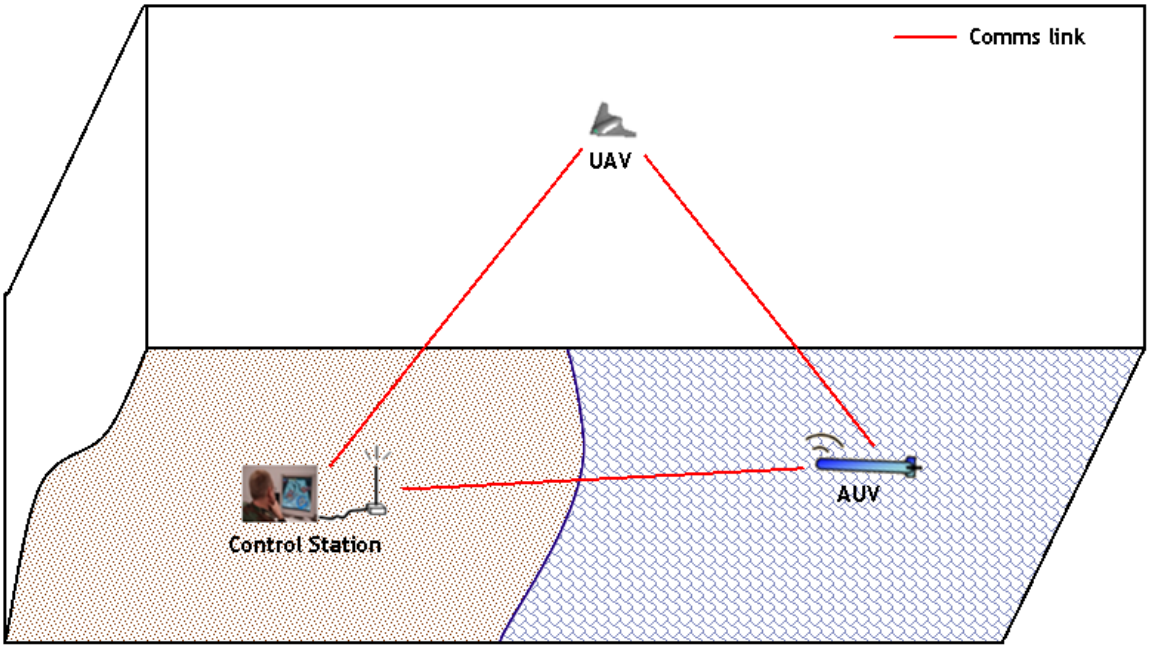

Figure 26 – Toy example mission overview

To better understand what this concept is and how it could be applied to an operational scenario, consider a “toy example” in which a system is composed by the main physical entities: UAV, AUV and a Control Station (Figure 26). The following configurations are assumed:

- UAV – simple manoeuvre controller (CA) and Wi-Fi communication system (PE)
- AUV – simple manoeuvre controller (CA), Wi-Fi communication (PE) and water sample system (PE)
- Control Station – Wi-Fi communication system (PE)

For this system, it would be easy to define a possible state characterization for each entity based on theirs configurations. As an example, some possible states for each of the entities, and their components, are presented in Table 1Table 2.

These states representations are simplified, as it would be possible to have a more detailed state if at each possible state value, some details were added. For example, if the “Goto” value for the manoeuvre executing state could have two sub-values: “from” and “to”, that would specify from and to where the vehicle is travelling.

UAV and AUV entities have a very similar characterization. Both could be idling or executing an operation, communicating with another entity or idling and executing a “Loiter” or a “Goto” manoeuvre. AUV has also another component, the water sample system that could be sampling or idling. The control station could, either, be idling or requesting tasks and communicating with other entities or idling.

Assuming an operational mission, at the toy example scenario, where an operator at the control station wants a water sample of a specific area, a possible executing approach would be:

- a command would be sent to the AUV with waypoints where to sample and after that, a communication waypoint where it should go and send the information gathered to the UAV
- the UAV would be commanded to go to the communication waypoint, wait there until it receives the information from the AUV and then go back to the control station.

Table 1 – Entities and components state

Entity	Entity State	Comms State	Manoeuvre Executing	Sensor State
UAV	{Idle, Executing}	{Idle, Communicate}	{Goto, Loiter}	-
AUV	{Idle, Executing}	{Idle, Communicate}	{Goto, Loiter}	{Idle, Sample}
Control Station	{Idle, Requests}	{Idle, Communicate}	-	-

Give this mission, at some point the four states shown in Figure 27 would occur. As presented in Table 2, at each of that states, it is possible to see what is happening in the world by gathering the entities state, as well as theirs components state.

Although this representation would provide a system state based on each entity state, it would not illustrate what really is happening, in terms of a composed operation. As said before, the composed operations problem arises from this point, and it will be presented in the next section.

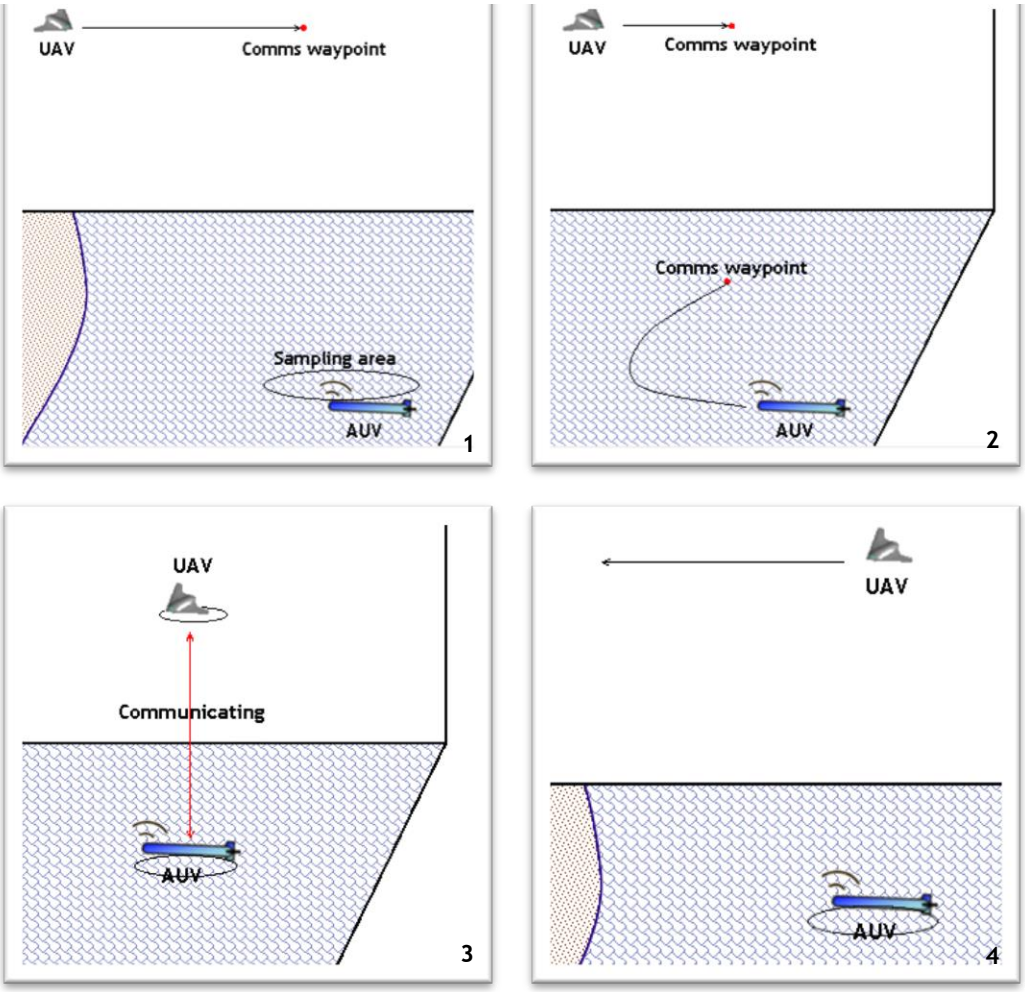

Figure 27 – Four states of the toy example mission

Table 2 - Entities and components state at four different states

State	Entity	Entity State	Comms State	Manoeuvre Executing	Sensor State
S1	UAV	Executing	Idle	Goto	-
	AUV	Executing	Idle	Loiter	Sample
	CS	Request	Idle	-	-
S2	UAV	Executing	Idle	Goto	-
	AUV	Executing	Idle	Goto	Idle
	CS	Request	Idle	-	-
S3	UAV	Executing	Communicate	Loiter	-
	AUV	Executing	Communicate	Loiter	Idle
	CS	Request	Idle	-	-
S4	UAV	Executing	Idle	Goto	-
	AUV	Idle	Idle	Loiter	Idle
	CS	Request	Idle	-	-

4.2.2 Composed Operations Specification

As said earlier, a new model to define composed operations is to be developed, that in addition to give advantage on large-scale systems state capture, it provides specific system entities with the planning knowledge to achieve them.

This model should be defined as a network of service providers [22] that together offer the possibility to establish composed operations, and provide additional features to a system. It should arise from the coordination and combination of system resources in particular situations. The ability to specific system entities identify what composed operations the system can offer, should derive from the composed operations specification, which should follow a design specification. This design specification should be enough detailed, not to compromise the achievement of a composed operation and as a result, provide new features to the system, but at the same time not too restrictive, in order to simplify its formation at any time.

4.2.2.1 Composed Operations Design Specification Requirements

Along this work a composed operation refers to the combination of single operations in order to provide additional features and capacities to a heterogeneous multi-tasking system. A single operation is defined by being a sequence of specific pre-defined actions in order to achieve the operation goal.

Again considering the example mission, described in the previous section, a composed operation could be specified as being a “Sampling” operation. The composed operation would have two different operations, “Relay” and “Gather”, to be executed by different entities. As said before, an abstract concept is an idea, type of thing, that have been condensed from

concrete realities, and so in a composed operation, the “Sampling” operation and its sub-operations can be modelled as abstract entities. In a simple way, if these abstract entities were associated with the entities models and, somehow, transformed into details of the entity states values, it would add more description to the system state. For example while in the four states of Figure 27, the entities states would be as in Table 3.

Although at this point the composed operations model would be just an approach to solve the cooperative operations problem in the system state capture, it would be also desired that it provided systems with the capabilities to trigger, achieve and output plans for them, when possible and requested. This would be possible if, as in [22], each system entity had the knowledge of how to execute the operations, it could provide, and some of these entities had the knowledge of how to achieve the composed operations.

Formally, a composed operation requester would be a system entity with the specification knowledge and that would have control over other entities, so that it could send plans, or objectives, and supervise their execution. From the design specification it would be possible to restrict the entities likely to be assigned to a single operation, as each entity would have an identifier (the abstract concept associated to the characterization) that confirmed the ability to perform it or not.

Table 3 – Entities state with values properties

Entity	Entity State
UAV	Executing(Sampling(Relay))
AUV	Executing(Sampling(Gather))
Control Station	Requests(Sampling(Relay(UAV)),Sampling(Gather(AUV))

Chapter 5

Approach

In the previous chapter, the problem was defined and two main sub-problems emerged:

- to identify and develop a model to the system entities
- to specify and develop a model to define composed operations in a system

It was described that having a well-defined characterization for a system main physical entities and their components is a crucial stage. After that, the composed operations model concept and the main features it should provide were also described.

After specifying the models to the system entities, and to the composed operations, it is desirable to somehow assemble the models. With a model for the whole system, the next goal is to implement it with a framework capable of building planners with a modelling approach.

5.1 Overview

Recapping the problem, there were two main issues to solve: the specification and development of a model to define composed operations and the specification of a model for the system entities. It was also desired to provide a planning method that would take advantages from knowing the system state and what composed operations it could provide.

The first step, on this approach, was to specify the models to the system entities and to the composed operations. After, it was desirable to formalize a model that would combine both, in order to be possible to develop a planner capable of implementing it. To solve the implementation problem, an appropriate platform should be chosen, so that it can output some kind of planning.

From what was learnt in section 3.1.3, it was decided to develop generic structures to model the system entities and the composed operations. The structures are object oriented,

being each object characterized by attributes, actions and some possible relationships with other objects. At Figure 28, the physical entities concept architecture is presented, following the characterization architecture from Figure 25.

The composed operations concept architecture, as is shown in Figure 29, is only composed by abstract entities objects. This was done because, as it was pointed out in section 3.1.4, networks topologies defined through abstract capabilities can be applied to various types of vehicles, even though the way of implementing it is different.

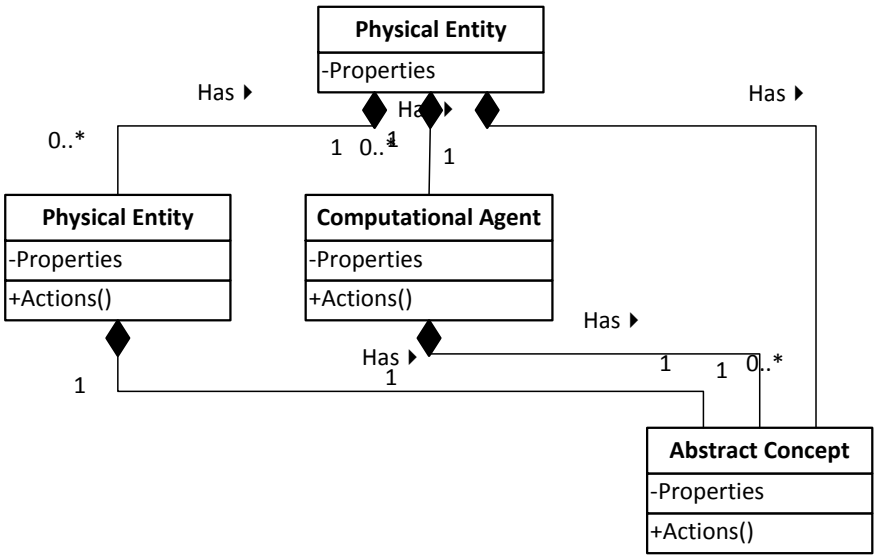

Figure 28 – Physical entity concept architecture

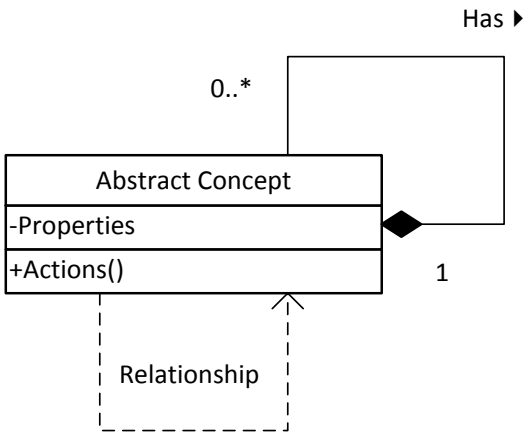

Figure 29 – Composed operations concept architecture

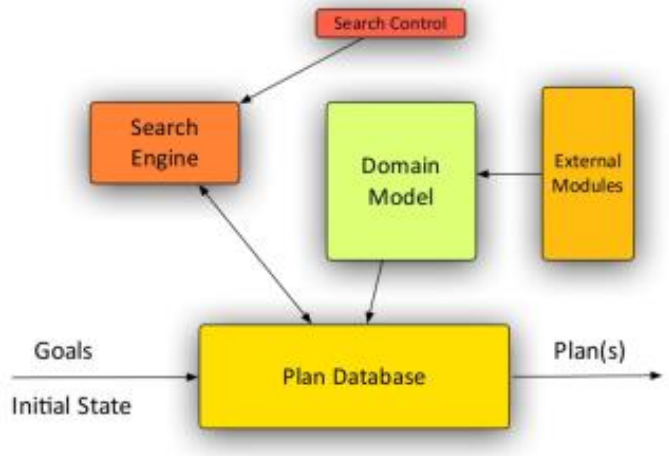

Figure 30 - A general architectural block diagram for an AI based Planner [38]

For the model implementation, it was proposed by LSTS, to use EUROPA (see section 3.3) because it would satisfy most of the problem requirements. One of EUROPA's key development goals is to streamline the process of integrating advanced planning, scheduling and constraint reasoning into an end-user application [33].

As seen in Figure 30, EUROPA uses a domain model, together with initial conditions and goals in order to output plans [38]. This means that a single planner can be applied to different systems and problems if different models and goals are provided [29].

Briefly, if a system model for the whole problem is formed, it can be applied together with initial conditions, which ideally would be the system state at the planning desired time, and goals in order to achieve composed operations and output plans for it.

5.2 System Specification

5.2.1 Physical Entities Model

As said in the problem chapter (Chapter 4), the properties considered to be crucial, when capturing the entity state, were to be identified based on the needs of experienced operators. After some meetings and discussions on the problem, the following properties were identified, as fundamental, on each main physical entity and its components. Firstly, the vehicles properties are going to be presented and, afterwards, the control stations properties.

5.2.1.1 Vehicles Model

Following Figure 28, the vehicles architecture is as presented in Figure 31. The objects will be explained bellow:

- Vehicle (Physical Entity):

  - Id – an unique identification of the vehicle
  - Type – an identifier of the vehicle type
  - Payload – all the payload the vehicle has
  - Abstract Concept - This concept will be approached in section 5.3
  - Movement Restrictions – movement restrictions as maximum speed the vehicle can travel
  - Operational autonomy – maximum operational autonomy
- Vehicle operational state (associated with the Vehicle object):
  - Control Station - control station that supervises the vehicle
  - Operation Time – time that has passed since the vehicle is operating
- Payload (Physical Entity). For each payload system the vehicle has an object associated with it. The payload object has as attributes:
  - State – the state attribute can have the values:
    - Assigned – the service the payload is assigned to (will be detailed section 5.3)
    - Unassigned – if the payload is not assigned
- Payload Actions (associated with the Payload object). For each payload system the vehicle has an object associated with it that maps its actions. The payload actions object has as attributes:
  - Action – the actions the payload can execute

  And as operations:
  + Execute – accepts as inputs an action and a location, and triggers the payload controller to execute it
- Communication (Physical Entity). The communication system for each vehicle has as attributes:
  - Action – the communication actions the vehicle can execute

  And as operations:
  + Communicate – accepts as inputs an action and an entity id, and activates the communication
- Navigator (Computational Agent). In this approach it is considered that a vehicle has a navigator that controls the vehicle movements. The navigator has as properties:
  - Manoeuvres – the manoeuvres that the vehicle can execute

  And as operations:
  + Navigate – accepts as inputs the actual position, the future position and the manoeuvre to be executed, and triggers the navigation controllers
- Navigator State (associated with the Navigator object):
  - State - the state attribute can have the values:
    - Underwater – if the vehicle is underwater

- Abovewater – if the vehicle is above water

- Abstract entities: This concept will be approached in section 5.3

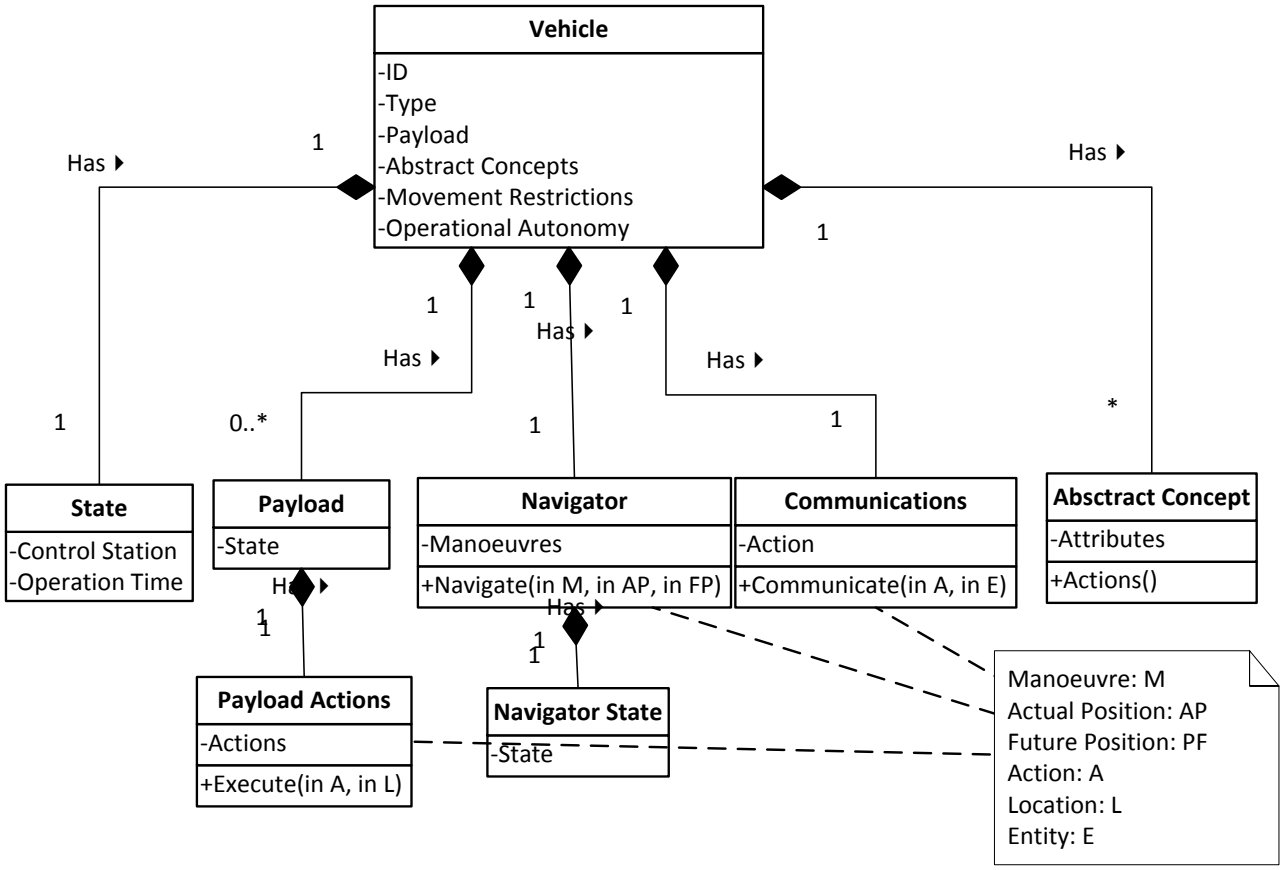

Figure 31 – Conceptual class diagram for vehicles model approach

5.2.1.2 Control stations Model

Again following Figure 28, the control stations architecture is as presented in Figure 33. The objects will be explained bellow:

- Control Station (Physical Entity):
  - Id – an unique identification of the control station
  - Cooperative configurations – all the cooperative configurations that the control station can require
  - Position – the control station position
- Communication (Physical Entity). The communication system for each control station has as attributes:
  - Action – the communication actions the vehicle can execute

  And as operations:
  + Communicate – accepts as inputs an action and an entity id, and activates the communication
- Abstract entities: This concept will be approached in section 5.3

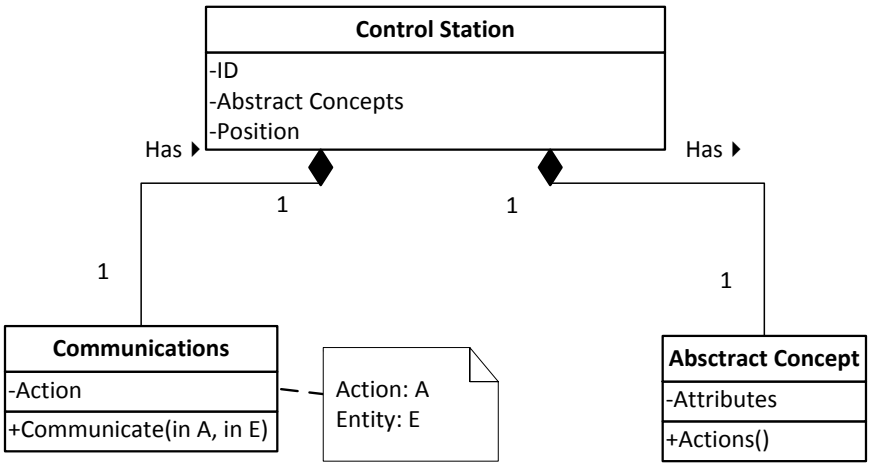

Figure 32 – Conceptual class diagram for control stations model approach

5.2.2 Composed Operations Model Specification

Along this work, a single operation will be called out as a Service and the combination of Services to compound a composed operation will be called out as a Composed Service. Following Figure 29, the Service architecture is as presented in Figure 33. As shown, Composed Services consist of two main entities: Services and Service Steps. The Composed Service, Service and Service Steps concepts can be identified as abstract entities. Composed Services are used to map real world composed configurations, Services are used to map the roles of each agent in the operation, and Service Steps are used to map real world actions from the agents. This means a Composed Service can be composed by several Services, being each Service composed by several Service Steps.

The objects specification will be explained bellow:

- Service attributes are classified as goals

Specifying these goals is an important step when creating a Service model because when requiring a Service they should be defined by the requester.

- Composed Service attributes are also classified as goals

They inherited from the Service attributes. When a Composed Service is created its goals will be the combination of the single Services goals. Composed Service goals can also derive from the Service assignment (will be detailed in section 5.3).

- Service Step attributes are:
  - specification of the manoeuvre, payload and/or communication action to execute
  - specification of the action goal, which are inherit from the Service goals

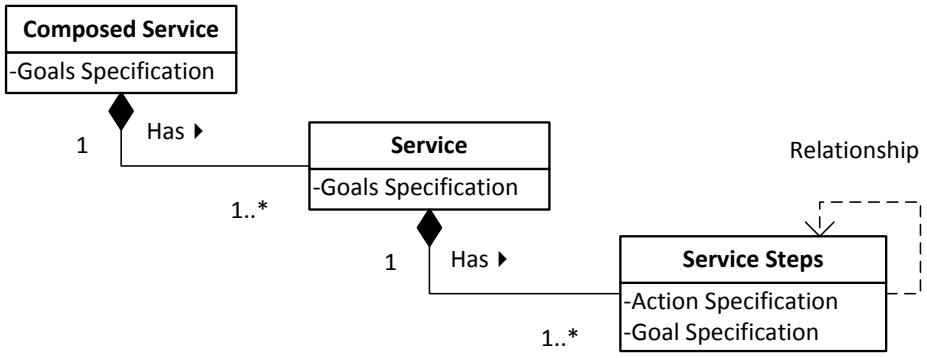

Figure 33 – Conceptual diagram for a service model

For a clear look on the model specification an example is presented. Considering, again, the example mission presented in the problem chapter (Chapter 4), and as said in section 4.2.2.1, a Composed Service could be specified as "Sampling", specified by two different Services, "Relay" and "Gather". The Services Steps executing sequence was built considering the executing approach and states described. A possible design to this composed operation, considering the service conceptual model, is presented at Figure 34.

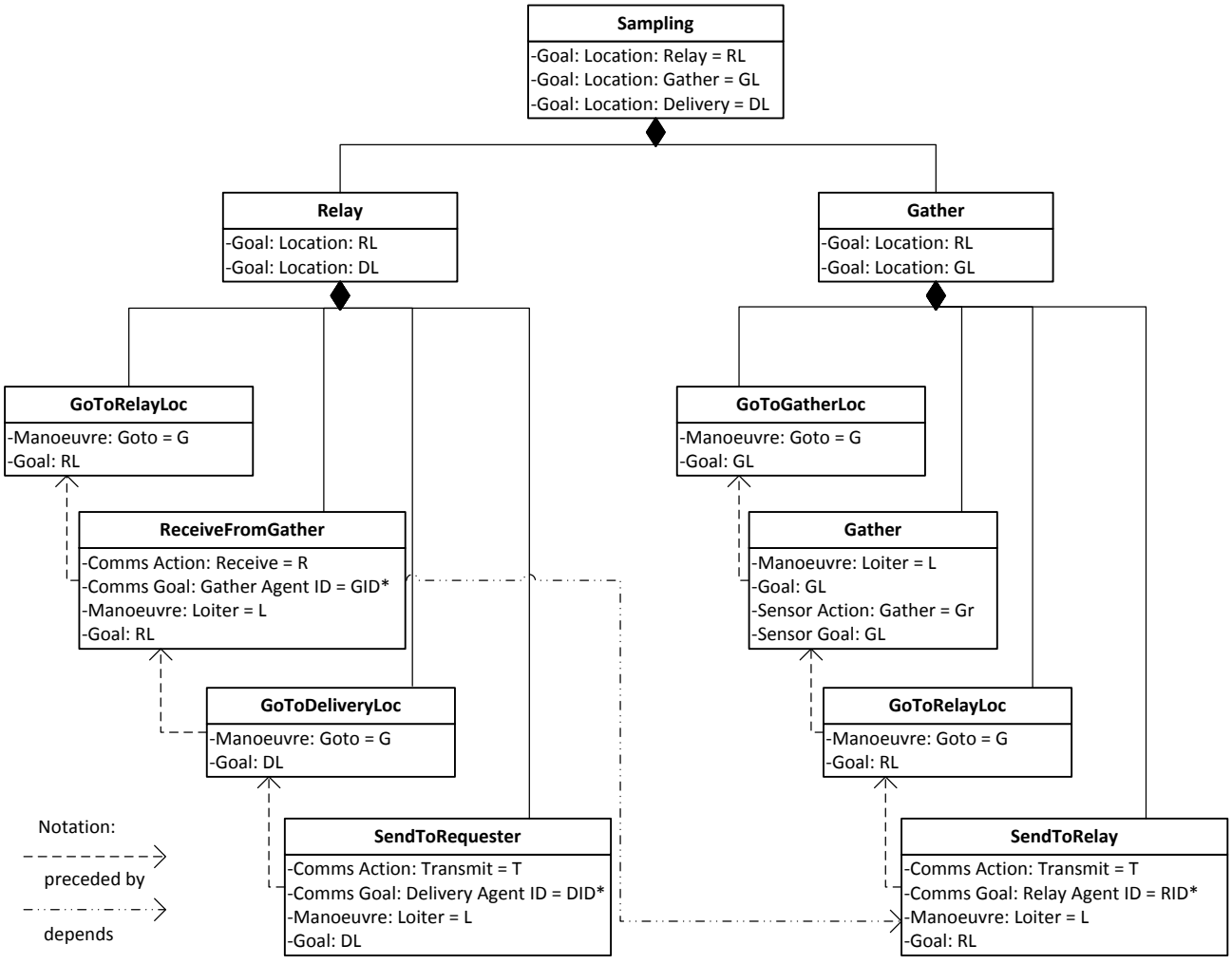

Figure 34 – Sampling service conceptual diagram

Looking at this design approach, it is possible to understand the expected execution sequence, of each Service executer. It is also understandable that the achievement of the Composed Service is only successful if all the Services that specified it are accomplished, and obviously, a Service is only successfully accomplished if the Service Steps execution sequence has also been accomplished.

Composed Service Property 1: The achievement of a Composed Service goal is only successful if all the Service goals that specify it are accomplished, and for that, the execution of the Services Steps sequence has to be accomplished as well.

As in some task allocation works reviewed in section 3.1.2, it can be stated that the Services Steps in this composed model are coupled by precedence and order constraints.

5.3 System Model

As seen in the previous section, 5.2, and in the problem chapter (Chapter 4), abstract concepts are a key element on an entity characterization for defining a system state, when operating advanced missions. Along with this, abstract concepts also appear as an important element on the composed operations model specification, as they are used to abstract composed operations, operations and their steps execution sequence. At Figure 35, it is possible to see an approach on a joint model, where the composed operations model is applied onto the control stations and the vehicles model.

With this model, it will be possible to:

- Associate in the vehicles model the specification of how to execute Services
- Associate in the control stations model the specification of how to achieve Composed Services
- Understand the composed configuration and its participants
- Get a detailed system state

Some important issues, on the model, have to be highlighted:

- A Composed Service is composed by Services that should be assigned and executed by vehicles
- The formation of the Composed Service is only successful when all the Services are assigned to at least one vehicle (**Composed Service Property 2**)

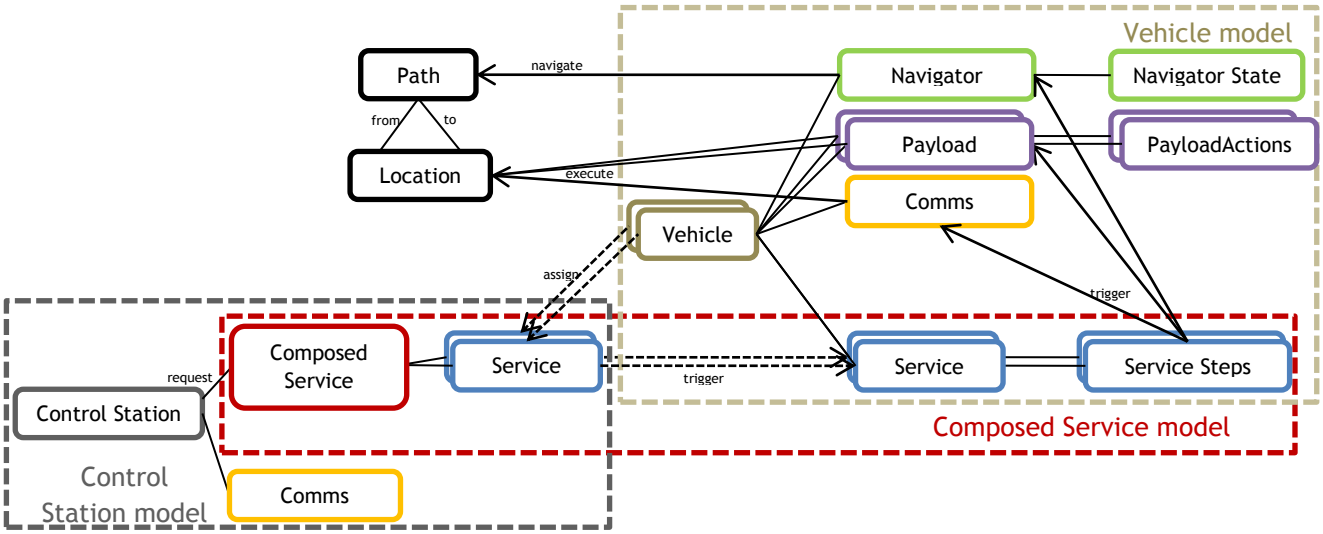

Figure 35 – System model concept

- The execution of the Composed Service only starts when the formation of the Composed Service is achieved (**Composed Service Property 3**)

Only after all the Services are assigned to at least one vehicle, and therefore the Composed Service is formed, their execution can start. This requires the addition of a state attribute to the Composed Service and Service concepts, on the Composed Service model (Figure 34). The state attribute describes if the Service is assigned or not to a vehicle, and when all the Services are assigned the Composed Service is considered to be assigned as well. After becoming assigned the Composed Service state becomes executing and triggers the Services state to executing as well.

- The Service concept has to be modelled in both the vehicle model that execute it and the control station that requests the Composed Service that needs it

Although the Service is modelled in both the models, in the control station model, the Service Steps execution sequence does not have to be modelled. This is, from the control station view it is only important to know what Services a Composed Service needs, and therefore the Composed Service and the Service concepts manage the assignment, executing and completion of a Composed Service. So, the state attribute, both on the Composed Service and Services, has the following values: Unassigned; Assigned; Executing; Completed. The Services state value has also an attribute that identifies the vehicle that has been assigned to them.

Figure 36 shows what has been described.

- Service Steps trigger the controllers that will execute the manoeuvres, which will navigate paths between two locations, payload and/or communications actions to be executed at a location

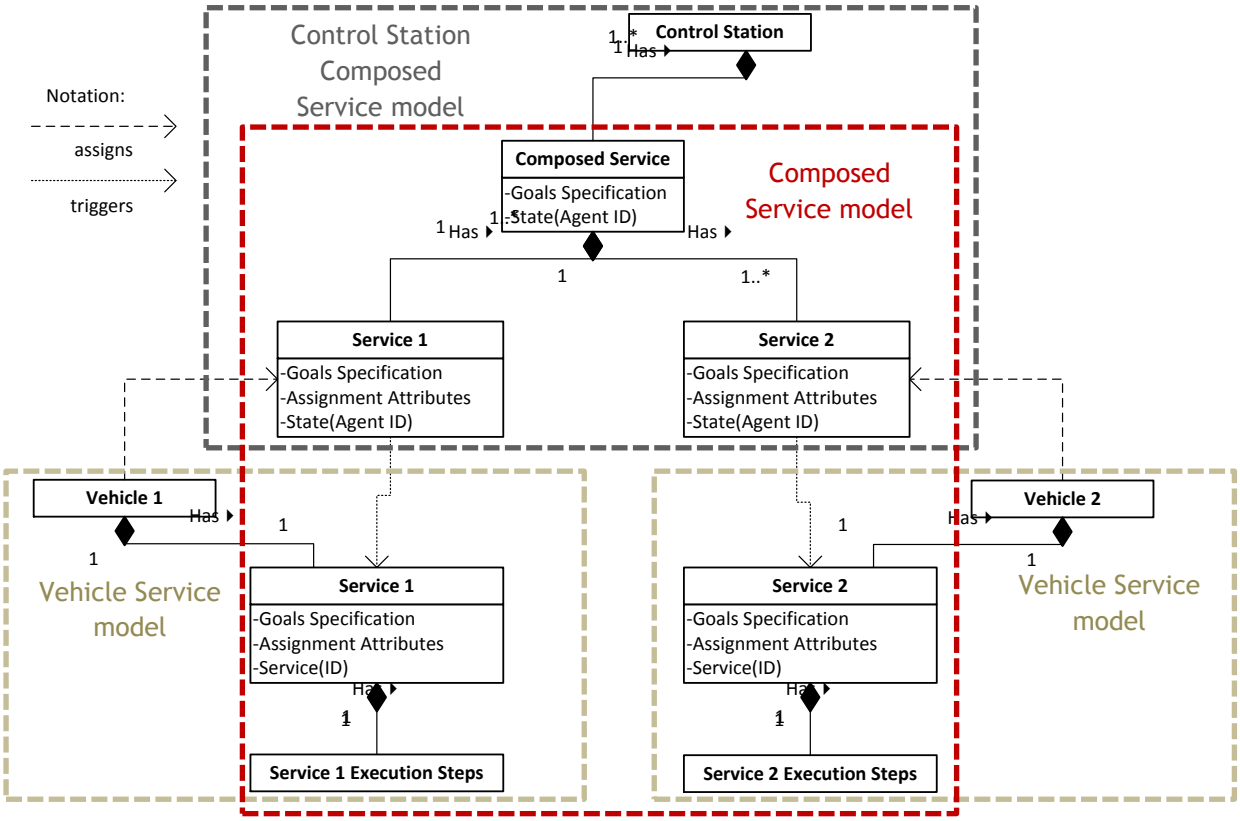

Figure 36 - Composed Service and Service conceptual model applied to Control Station and Vehicle conceptual model

This does not have to be done, necessarily, by means of a nested controller, on the Service component, but by sending objectives to the components that are in charge of controlling the desired component. For example, at Figure 34, in the "GoingToComsLoc" step from the "Relay" Service, it has to send the objective "Goto(Relay Location)" to the navigator component that will drive the vehicle through a "Goto" manoeuvre to the location "Relay Location". The step is considered complete as soon as the navigator indicates that the actual location is the one sent as objective, "Relay Location", and this would trigger the following step, "ReceivingFromGather".

- When a Service Step has a dependency on another Service Step, it means the "master" Service Step triggers the end of the "slave" Service Step

Once the vehicle, executing the "Relay" Service, starts the "ReceivingFromGather" step, it waits for a communication link from the vehicle executing the "SendingtoRelayer" step. This vehicle establishes the link and, after transmitting the data, the step is over. At this point, the "ReceivingFromGather" step ends.

5.4 Europa Application

Developing EUROPA applications is a design job that involves multiple iterations from an initial concept model to an actual NDDL encoding. A good approach is to gradually build up a domain description adding more detail methodically [29]. This approach was inspired by the approach provided at [29], where a simple planetary rover application is provided.

5.4.1 Application Domain Analysis

The first stage was to draw a concept map of the entities in the application domain and their relationships. This has been done in section 5.3, where a concept map for the system application domain was presented (Figure 35). The concept map focused on modelling the main entities, control stations, services and vehicles, and their interactions.

The next step was to identify the entities called timelines in the model application concept diagram that describe changes in the state of the system:

- **Composed Service** – manages the assigning, execution and conclusion of the Composed Service
- **Service (Control Station)** – manages the assigning, execution and conclusion of the Service
- **Service (Vehicle)** – manages the execution sequence
- **Service Steps** – manages the vehicle execution of the Service Steps
- **Vehicle** – controls the assignment of a vehicle to a Service and a Composed Service
- **Navigator** – controls the vehicle's movements
- **Navigator State** – manages the state of the vehicle navigation
- **Communications** – controls the communication's actions
- **Payload** – controls the payload's actions
- **Payload State** – manages the assignment of a vehicle payload to a service

The next step is to identify the states, predicates, which each timeline can be in. Some of these have already been approached in the previous sections. Figure 37 shows the set of predicates identified for each timeline, as described below:

- **Composed Service** – a Composed Service can be Available, Requested, Executing or Completed

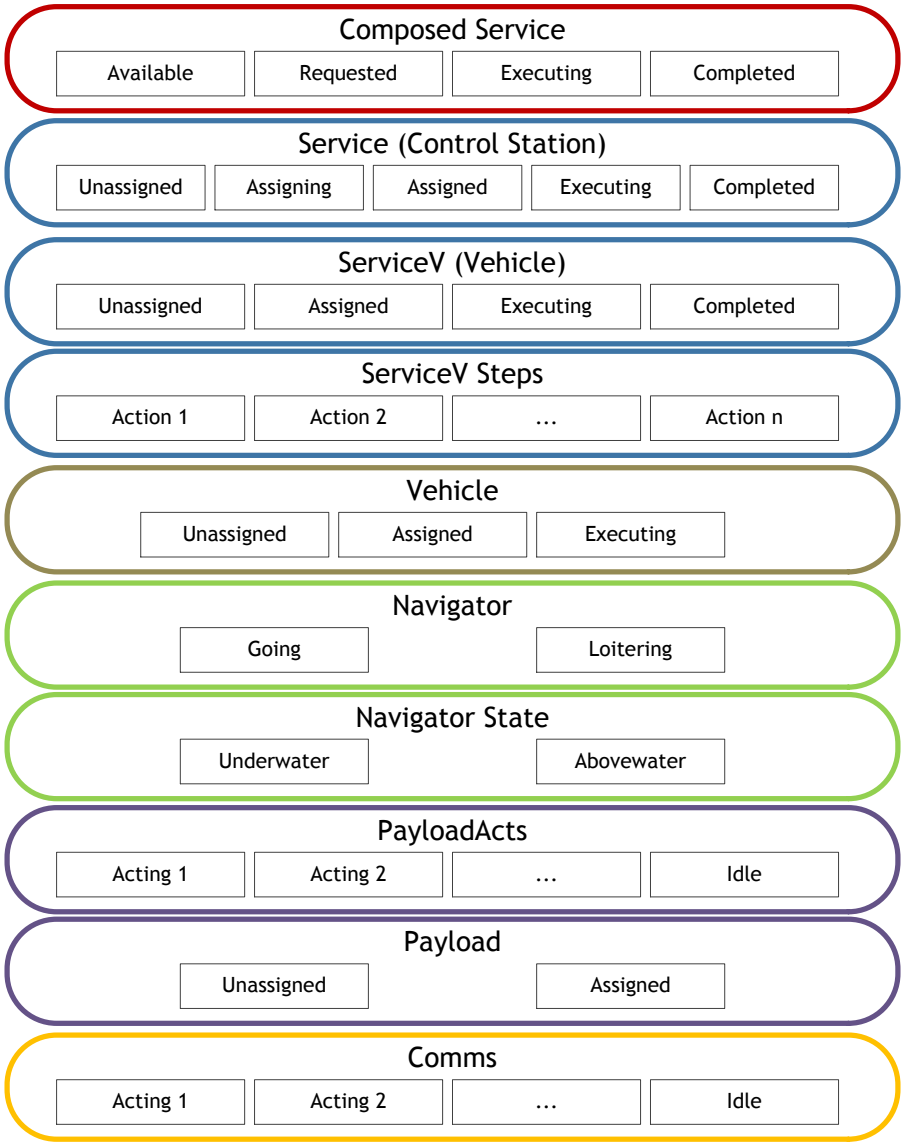

Figure 37 - Initial timelines and predicates

- **Service (Control Station)**- a Service, on the Control Station model, can be Unassigned, Assigning, Assigned, Executing or Completed
- **Vehicle State** - a vehicle can be Unassigned, Assigned or Executing a Service
- **Service (Vehicle)** - a Service, on the Vehicle model, can be Unassigned, Assigned, Executing or Completed
- **Service Steps** - the steps sequence the vehicle will perform when assigned and executing a Service. Each predicate matches a step
- **Navigator** - a vehicle either is going to a location or maintaining at a position (loitering)

- **Navigator State** – a vehicle can be underwater or above water (abovewater)
- **Communications** – a vehicle or a control station can be communicating with other vehicles or idling. Each predicate matches a different communication action
- **PayloadActions** – a vehicle payload can be either at use or idling. Each predicate matches a different payload action
- **Payload** – a vehicle payload can be assigned to a service or unassigned(free to be assigned in a Service)[1]

[1]This will become important if vehicles execute different Services at once, where no payload restrictions conflicts can occur, and so a payload can only be assigned to one Service.

The next and final step is to detail the properties of the predicates and the constraints between them. The use of constraints between the predicates has the function to define acceptable behaviour for the system and to disallow the unacceptable one, as it will be seen next. The predicates properties are going to be detailed carefully in section (5.4.2), but they are no more than the attributes already defined in the models (sections 5.2.1,5.2.2 and 5.3).

Figure 38 shows the constraints for a Composed Service request, executing and completion. The temporal relations between the timelines map the service model specification and properties, sections 5.2.2 and 5.3:

- For a Composed Service to be Executing all the Services have to be Assigned
- A Service has to be Assigned to a Vehicle Unassigned to a Service
- The assignment of the Vehicle to a Service, in the Control Station model, will trigger the state of the Service, in the Vehicle model, to Assigned as well, and it is required that the Service is Unassigned
- When all Services are Assigned, the Composed Service can start Executing and this triggers the Services state, in the Control Station model, to be Executing, and consequently this triggers the Vehicle state and the Services state, in the Vehicle model, to Executing as well
- As soon as the Service state goes Executing, the Service Steps sequence will start
- When the last step is performed, the Service Executing state will end and this triggers the Vehicle state to Unassigned, and consequently this triggers the Service state, in the Control Station model, to Completed

- When all Services are Completed, the Composed Service becomes Completed as well

The Vehicle model also has some constraints among their components, as shown in Figure 39:

- The Communications Actions have to occur not only while the Vehicle is Executing a Service, but also while the Vehicle is Abovewater.

This is important to underwater vehicles because while submerged they cannot communicate.

- Actions succeed and precede the Idle state

In what concerns the Navigator:

- The Going manoeuvre also has to occur when the Vehicle is Executing a Service, since the orders to move come from the Service Steps.

- A Going manoeuvre is preceded by a Loitering manoeuvre, that represents the act of being at a fixed location, and meets another Loitering, but in a different location.

- The Navigator state derives from the Vehicle position. When being at a position (only for underwater vehicles), if the depth becomes below zero, the state becomes Underwater. The opposite triggers the Abovewater state. Each state follows the other.

The Vehicle Payload constraints are shown in Figure 40:

- When a Vehicle is Assigned to a Service, as explained before, specific Payload, derived from the Service specification, can be required to be Assigned as well. In this circumstance the Service, in the Vehicle model, will trigger the assignment of the specified Payload.

In this circumstance the specified Payload should become Assigned as soon as the Service becomes Assigned.

- The Payload has to be Unassigned before becoming Assigned

- Every PayloadAction predicate has to occur while the Payload is Assigned, and therefore the Vehicle, to a Service and every action succeeds and precedes the Idle state

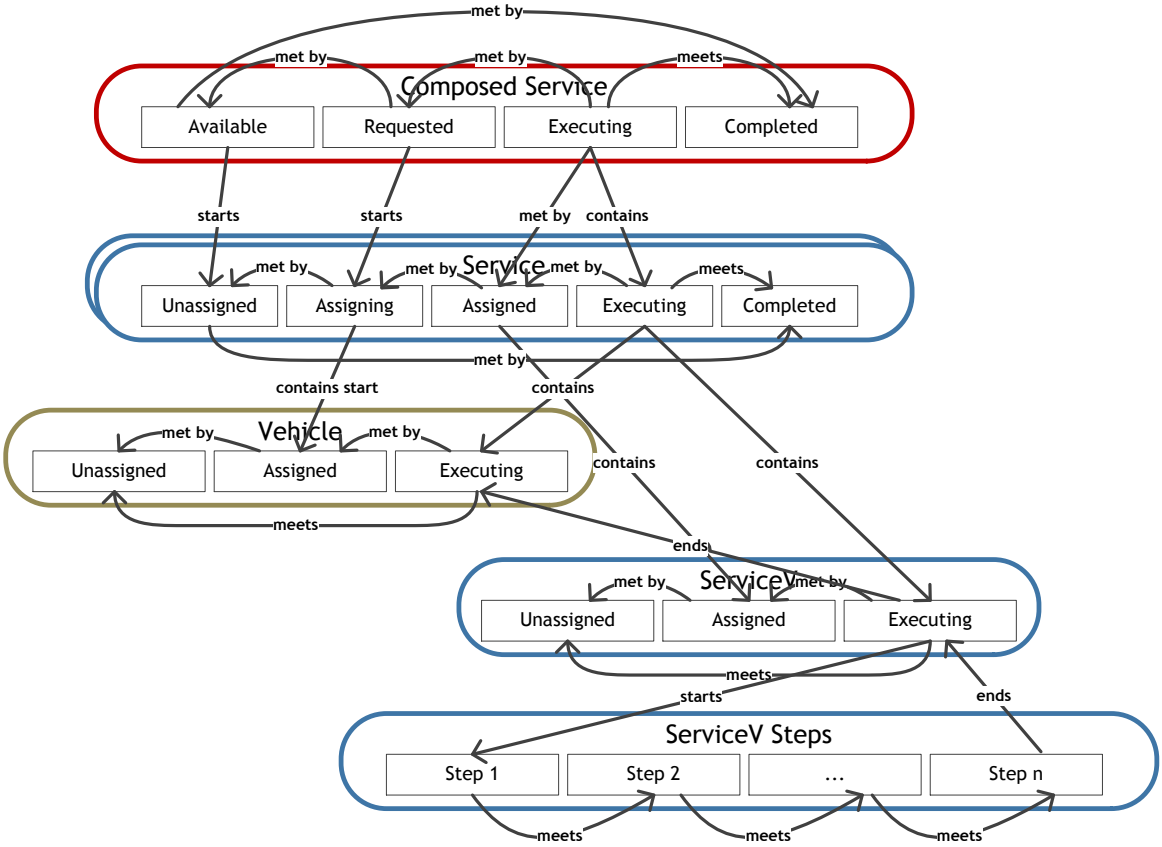

Figure 38 - Timelines and predicates with transitions between them - Composed Service request, executing and completion constraints

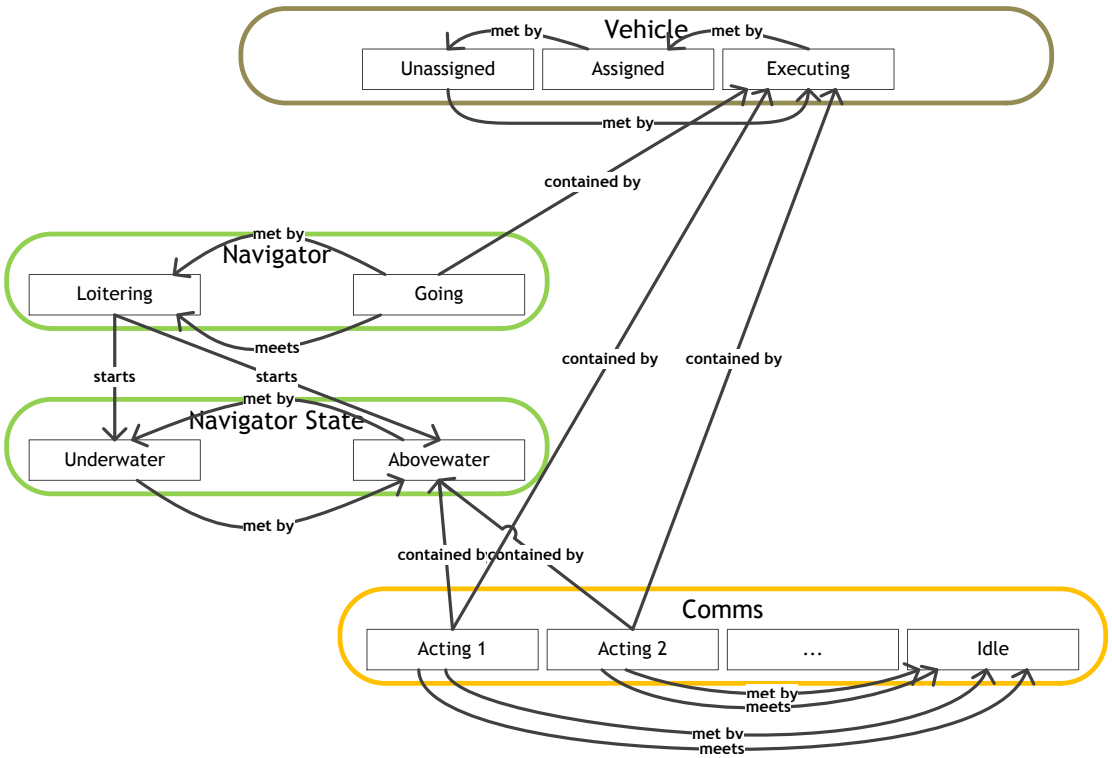

Figure 39 - Timelines and predicates with transitions between them - Vehicle model constraint

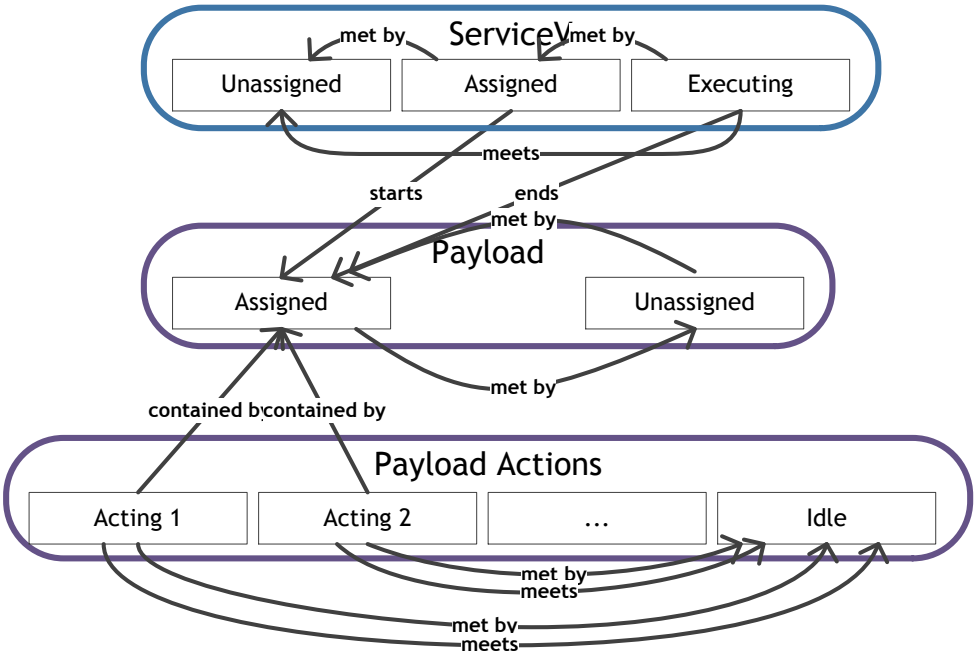

Figure 40 - Timelines and predicates with transitions between them: Payload assignment and actions

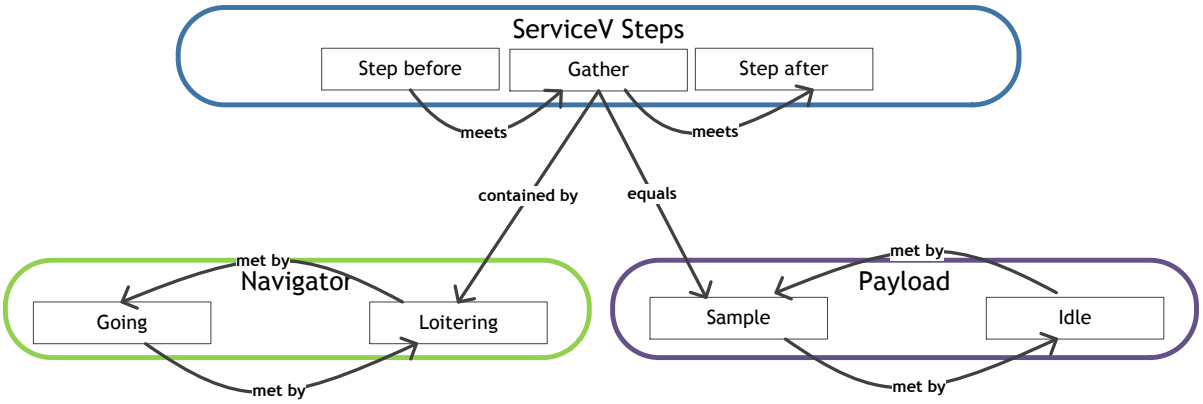

Figure 41 – Timelines and predicates with transitions between them: constraints for a Gather Step, one possible step of a Service Steps execution sequence

Figure 41 shows an example of the constraints between a Service Steps timeline predicate (step Gather) and other timelines predicates that are related to the step. The contained by constraint specifies that the Vehicle must be Loitering, at a specific Location, for the duration of the step in order to be able to Sample at the specified Location. The constraint with the Payload timeline shows that when the step starts the Payload action must begin and when the action ends the step also ends, being the Gather step duration equal to the action of Sample.

5.4.2 NDDL Encoding

At this section, each model file will be stepped thought and explained how it derived from the analysis made. The encoding presented is the generic approach made on the models and a real application may need some changes. In Chapter 6, an example application is presented.

5.4.2.1 Defining the Vehicle Model Timelines

Encoding the components of the vehicle, first the *Navigator* is presented. It manages the *Vehicle* navigation and has a *NavigatorState* attribute. This class contains the two predicates identified earlier. The *Loitering* predicate models the concept of the vehicle being at a particular *Location Loitering*. The *Going* predicate models the concept of moving between *Locations*. The *neq* construct is a constraint that ensures the vehicle does not attempt to go to the actual position.

```
1   class Navigator extends Timeline
2   {
3           Vehicle vehicle;
4           NavigatorState state;
5
6           Navigator(Vehicle v) {
7                   vehicle = v;
8                   state = new NavigatorState(this);
9           }
10
11          predicate Loitering  {Location at;}
12
13          predicate Going { // Vehicle may be going between two locations
14                  Location from;
15                  Location to;
16                  neq(from, to); // prevents vehicle from going from a location straight back to that location
17          }
18  }
```

List 1 – class Navigator

The *NavigatorState* timeline has two predicates declared but have no accompanying logic. The only constraint is that they occur on a single timeline, so cannot overlap.

```
1   class NavigatorState extends Timeline
2   {
3           Navigator nav;
4
5           NavigatorState(Navigator n)  {
6                   nav = n;
7           }
8
9           predicate Underwater {}
10          predicate Abovewater {}
11  }
```

List 2 – class NagivatorState

Payload timeline itself details the assignment of the *Vehicle Payload* to *Services*. It has three attributes: the *Vehicle* the *Payload* belongs to, the type of *Payload* it is (*pType*) and

contains the *PayloadActs* timeline. The *Assigned* predicate has the parameter *pType* that specifies the Service the payload is assigned to.

```
1   class Payload extends Timeline
2   {
1           Vehicle vehicle;
2           pType ptype;
3           PayloadActs actions;
4
5           Payload(Vehicle v,  pType st) {
6                   vehicle = v;
7                   ptype = st;
8                   actions = new PayloadState(this);
9           }
10
11          predicate Unassigned {}
12          predicate Assigned { tType ttype;}
13  }
```

List 3 – class Payload

The *PayloadActs* timeline details the management of the vehicle's *Payload* actions. It has an attribute that maps the *Payload* to which the actions refer to. The predicates map the possible *Payload* actions the vehicle has, being a specific *Location* a parameter. The predicate *Idle* maps the state when the *Payload* is not acting.

```
1   class PayloadActs extends Timeline
2   {
3           Payload payload;
4
5           PayloadActs(Payload p) {
6                   payload = p;
7           }
8
9           predicate Idle {}
10          predicate Action 1 { Location at;}
```

List 4 – class PayloadActs

The *Communications* timeline is very similar to the *PayloadActs*. The difference is that the attribute maps the V*ehicle* to which the *Communications* system belongs to and the actions parameters are a *Vehicle* and a *Location* specification.

```
1   class Comms extends Timeline
2   {
3           Vehicle vehicle;
4
5           Comms(Vehicle v) {
6                   vehicle = v;
7           }
8
9           predicate Idle{}
10          predicate Action 1 {
11                  Location location;
12                  Vehicle vehicle_to;
13          }
14  }
```

List 5 – Class Comms

Finally, the *Vehicle* class puts together all the components defined in this section and manages the assignment (*Assigned*) and execution (*Executing*) to Services. It has several attributes, among them the *Payload*, *Navigator* and *Communications* classes, and the *Vehicle* Services specification (*svType*). The *Assigned* and *Executing* predicates have the the parameter sv*Type* that specifies the *Service* the *Vehicle* is *Assigned* to and *Executing*.

```
1   class Vehicle extends Timeline //class vehicle
2   {
3               string id; //vehicle identification
4               vType type; //vehicle type
5               float maxVel; //vehicle max velocity
6               Payload payload; //vehicle payload system
7               Navigator navigator;    // Keeps track of vehicle's position
8               Comms comms; //vehicle communications system
9               svType svtype; //vehicle service types
10
11              Vehicle(string i, vType t, float mv, svType svt) {
12                      id = i;
13                      svtype = svt;
14                      maxVel = mv;
15                      navigator = new Navigator(this);
16                      comms = new Comms(this);
17                      payload = new Payload(this);
18                      ttype = tt;
19              }
20
21              predicate Unassigned {}
22              predicate Assigned { svType svtype;}
23              predicate Executing { svType svtype; }
24  }
```

List 6 – class Vehicle

Going through the *Vehicle* predicates, it is simple to understand that they are declared but have no accompanying logic. The only constraints are that they cannot occur at the same time and they have a defined order through which they must happen. The parameter *tType* that the *Assigned* and *Executing* predicates have is defined when the *Task* is *Assigned* (see List 21).

```
1   Vehicle::Unassigned
1   {
2               met_by(Executing);
3   }
4
5
6   Vehicle::Assigned
7   {
8               met_by(Unassigned);
9   }
10
11  Vehicle::Executing
12  {
13              met_by(Assigned);
14              meets(Unassigned);
15  }
```

List 7 – Predicates from Vehicle class

The *Navigator* predicates, as said before, are *Loitering* and *Going*. The first has just the precedent condition that it must be met by a *Going* manoeuvre, which means that for being at a *Location* the *Vehicle* had to go there. This predicate also manages the *NavigatorState*

timeline state by triggering one of the two states: *Underwater* and *Abovewater*. This is done by evaluating the *z* coordinate of the actual and last *Location*. If there is a change, comparing it with zero, it triggers the right state.

```
1   Navigator::Loiter
2   {
3               met_by(Going g);
4               location == g.to;
5               if(location.z >= 0 && g.from.z < 0) {
6                       starts(object.state.Abovewater);
7               }
8               if(location.z < 0 && g.from.z >= 0){
9                       starts(object.state.Underwater);
10              }
11  }
```

List 8 – Predicate Loiter from Navigator class

The *Going* predicate has three conditions: it is preceded by a *Loitering* manoeuvre, is followed by a *Loitering* manoeuvre and must happen while the *Vehicle* is *Executing* some *Service*. It is easy to understand that a *Going* manoeuvre has to be from one *Location* to another. The *Location* from where the *Vehicle* is *Going* must be the last *Location* the *Vehicle* was, and the *Location* where the *Vehicle* is heading, is going to be the next *Location*. The duration of the predicate is at least the maximum velocity of the vehicle versus the displacement. To the *Vehicle* to go to a *Location* it has to be triggered by a *ServiceSteps predicate*, and so this can only happen when the *Vehicle* is *Executing* a *Service*.

```
1   Navigator::Going
2   {
3               contained_by(object.vehicle.Executing);
4               met_by(Loiter _from);
5               meets(Loiter _to);
6               to == _to.location;
7               from == _from.location;
8               abs(to-from)*velocity <= duration;
9   }
```

List 9 – Predicate Going from Navigator class

As the *Vehicle* predicates, the *NavigatorState* predicates are declared because they cannot occur at the same time.

```
1   NavigatorState::Underwater
2   {
3               met_by(Abovewater);
4   }
5
6   NavigatorState::Abovewater
7   {
8               met_by(Underwater);
9   }
```

List 10 – Predicates from NavigatorState class

Going now through the *Payload* predicates, again it is simple to understand that they are declared but have no accompanying logic. The *Assigned* predicate has the *svType* parameter

that expresses the *Service* to which the *Payload* is *Assigned* to. Again this is defined when the *Service* is *Assigned* (see List 28).

```
1   Payload::Assigned
2   {
3           met_by(Unassigned);
4           meets(Unassigned);
5   }
```

List 11 – Predicate Assigned from Payload class

The *Actions* predicate, both in the *PayloadActions* class or in the *Comms* class, always succeeds and precedes the *Idle* state. The *Action* in the *PayloadActions* also has another condition: an action will only happen when the *Payload* is *Assigned* to a *Service*. As in the *Going* predicate, it has to be triggered by a *ServiceSteps predicate*.

```
1   PayloadActions::Action
2   {
3           contained_by(object.sensor.Assigned);
4           met_by(Idle);
5           meets(Idle);
6   }
```

List 12 – Action predicate from PayloadActions class

```
1   Comms::Action
2   {
3           met_by(Idle);
4           meets(Idle);
5   }
```

List 13 – Predicates from Comms class

5.4.2.2 Defining the Composed Service Model Timelines

The *Composed Service* model will be encoded bellow. As it was said before, four timelines were identified: *Composed Service* and *Service*, in the *Control Station* model; *ServiceV* and *ServiceSteps*, in the *Vehicle* model.

5.4.2.2.1 Composed Service and Service Timelines

Before presenting the *Composed Service* timeline itself, the *Service* timeline has to be introduced, as it is a sub-timeline of the first. The *Service* timeline is encoded in the *Service* class. The class has two attributes: *Composed Service* that maps the Composed Service to which the *Service* belongs to, and *svType* that specifies the type of *Service* it is. The second part of the class specifies the constructor, which defines how the attributes are initialized when a new instance is created. The last part contains the predicates, identified earlier, and its parameters. The *Assigning*, *Assigned* and *Executing* predicates include the specific *Vehicle* and Service type (sv*Type*) parameter.

```
1   class Service extends Timeline //class Service
2   {
3       ComposedService composedservice; //Keeps track of the Composed Service state
4       svType svtype; //Specification of the Service
5
6       Task(ComposedService s, svType svt) {
7               composedservice = s;
8               svtype = svt;
9       }
10
11      predicate Unassigned {}
12      predicate Assigning {
13              Vehicle vehicle;
14              pType ptype; //If there is a payload requirement
15          }
16      predicate Assigned {
17              Vehicle vehicle;
18      }
19      predicate Executing {
20              Vehicle vehicle;
21              Goals goals;
22      }
23      predicate Completed {
24              duration = 1;
25      }
26  }
```

List 14 – class Service (Service in Control Station model)

The *Composed Service* timeline itself details the supervision of the *Services* assignment and execution. It contains the *Service* timelines in its construction and the *Requested*, *Executing* and *Completed* predicates have the *Goals*, which will be distributed to the *Services*, as parameters. *Goals* are just generic properties that in a real implementation should take the form of a world object, e.g., if the real goal is to get to a location it would be like: *Location goal*.

```
1   class ComposedService extends Timeline //Class Sampling
2   {
3           Service service1; //Service 1 of the Composed Service
4           Service service2; //Service 2 of the Composed Service
5
6           ComposedService() {
7                   service1 = new Service(this,t_type); //Service specification
8                   service2 = new Service(this,t_type); //Service specification
9           }
10
11          predicate Available {}
12          predicate Requested { Goals goals;}
13          predicate Executing { Goals goals;}
14          predicate Completed { Goals goals;}
15  }
```

List 15 – class ComposedService

After defining the predicates, it is needed to specify the detailed constraints on each, starting with *the Available* predicate. The *ComposedService* is *Available* after being *Completed* and when a *ComposedService* becomes A*vailable*, the *Services* that specify it become *Unassigned*.

```
1   ComposedService::Available
2   {
3           met_by(Completed);
4           //When the Sampling is available the service that specify it are unassigned
5           starts(object.service1.Unassigned);
6           starts(object.service2.Unassigned);
7   }
```

List 16 – Available predicate from ComposedService class

The *Requested* predicate is similar to the *Available* predicate, but it is preceded by the A*vailable* state and starts the *Assigning* predicate of the *Services*. If a *Service* has a required *Payload* it should be passed as a parameter to the *Service Assigning* predicate.

```
1   ComposedService::Requested
2   {
3                                                   //ComposedService has to be Available to be Requested and when
            met_by(Available);                      //requested starts the assigning of the services
4           starts(object.service1.Assigning);
5           starts(object.service2.Assigning a);    //If the service has a required payload
6           a.ptype == ptype;
7   }
```

List 17 – Requested predicate from ComposedService class

Executing has two precedent conditions: the *ComposedService* has been *Requested* and the *Services Assigned*. The *Executing* goals and the *Payload* type are, obviously, the same as the *Requested* ones. The predicate contains the execution of the *Services* and the goals distributed to each derive from the *Requested* ones.

```
1   ComposedService::Executing
2   {
3           //ComposedService begins executing after being requested and after the services
4           //being assigned.
5           met_by(Requested requested);
6           requested.goals == goals;
7           requesetd.ptype == ptype;
8           met_by(object.service1.Assigned);
9           met_by(object.service2.Assigned);
10
11          //The ComposedService executing contains the services execution. The goals and the
12          //vehicles assigned are sent as parameters.
13          contains(object.service1.Executing service1);
14          service1.goals == goals;
15  }
```

List 18 – Executing predicate from ComposedService class

The *Completed* predicate is very simple. It is preceded by the *Executing* predicate and the *Completed* goals and *Payload* type are the same that were in the *Executing* predicate.

```
1   ComposedService::Completed
2   {
3           met_by(Executing executing);
4           executing.goals == goals;
5           executing.ptype == ptype;
6   }
```

List 19 – Completed predicate from ComposedService class

Inspecting now the *Service* predicates, the *Assigning* has two conditions: it is preceded by the *Unassigned* predicate and contains the start of the *Assigned* predicate from a *Vehicle* class. This *Vehicle* has to meet the *Service* specification and it is passed as a parameter to the *Vehicle Assigned* predicate.

```
1   Service::Assigning                              //Choose vehicle to assign in the Service
2   {
3           met_by(Unassigned);
4           contains_start(vehicle.Assigned assigned);          //Vehicle will be assigned
5           vehicle.ttype == object.ttype;          //Vehicle has to meet the Service specification
6           assigned.ttype == object.ttype;
7   }
```

List 20 – Assigning predicate from Service class

The *Assigned* predicate has the *Assigning* state as precedent condition and a parameter that maps the *Vehicle* that was *Assigned*. It triggers the *Service* in the *Vehicle* model (*ServiceV*) to be *Assigned*. The *ServiceV* has to be specified as the same as the *Service* being assigned.

```
1   Service::Assigned
2   {
3           met_by(Assigning assigning);
4           vehicle == assigning.vehicle;
5
6           ServiceV service;                       //The same service but in the vehicle model
7
8           service.vehicle == assigning.vehicle;   //Restricts the vehicles with the specified service  to
9           contains(service.Assigned assg);        // the assigned vehicle
10  }
```

List 21 – Assigned predicate from Service class

The *Execution* predicate has three conditions: it is preceded by the *Assigned* state and the vehicle that was assigned is the vehicle that is going to execute the *Service*; the Service execution contains the vehicle *Service* (*ServiceV*) execution and the goals are sent as parameters; after completion the *Service* state becomes *Completed*.

```
1   Service::Executing                              //Service execution with the vehicle assigned
2   {
3           met_by(Assigned assigned);
4           vehicle == assigned.vehicle;
5
6           contains(vehicle.Executing);
7
8           ServiceV service;
9
10          service.vehicle == assigned.vehicle;
11          contains(service.Executing exe);
12          exe.goals == goals;
13
14          meets(Completed);
15  }
```

List 22 – Executing predicate from Service class

The *Completed* predicate is just a temporary state with defined duration that precedes the *Unassigned* predicate.

```
1   Service::Completed                  //Service is completed after the executing has ended
2                                       // and should end  before the completion of the ComposedService
3   {
4           meets(Unassigned);
5   }
```

List 23 – Completed predicate from Service class

5.4.2.2.1 ServiceV and ServiceSteps timelines

The class *ServiceV* encodes the *Service* timeline in the *Vehicle* model. This class has two attributes: *Vehicle* and *Steps*. The *Vehicle* attribute maps the vehicle to which the *Service* belongs and the *ServiceVSteps* attribute maps the object that contains the steps sequence that specify the *Service*. The predicates, as identified earlier, are *Unassigned*, *Assigned*, *Executing* and *Completed*. The *Assigned* and *Executing* predicates include the parameter that specifies the *ComposedService* to which the vehicle *Service* is assigned to (*ComServiceType*). The *Executing* predicate has two additional parameters: *Goals*, which map the goals sent from the *Service* class in the *Control Station* model, and the identification of the *Vehicles Assigned* and *Executing* the other *ComposedService Services*.

```
1   class ServiceV extends Timeline
2   {
3           Vehicle vehicle;
4           ServiceVSteps steps;
5
6           ServiceV(Vehicle v)
7           {
8                   vehicle = v;
9                   steps = new ServiceVSteps(this);
10          }
11
12          predicate Unassigned {}
13          predicate Assigned {
14                  ComServiceType csvtype;
15          }
16          predicate Executing {
17                  ComServiceType csvtype;
18                  Vehicle vehicle_otherservices;
19                  Goals goals;
20          }
21          predicate Completed {
22          duration = 1;
23          }
24  }
```

List 24 – class ServiceV (Service in Vehicle model)

The *ServiceVSteps* timeline has just the *ServiceV* attribute that maps the *Service* to which the *steps* belong to. The predicate presented is just an example, as each *Service* is defined by different steps, and each one has different specifications in what concerns *Manoeuvre*, *Communication* and *Payload* actions, and *Goals*.

```
1   class ServiceVSteps extends Timeline
2   {
3           ServiceV servicev;
4
5           ServiceVSteps(ServiceV sv)
6           {
7                   servicev = sv;
8           }
9
10          predicate Action1
11          {
12                  Manoeuvre manoeuvre;
13                  CommsAct commsact;
14                  PayloadAct payloadact;
15                  Goals goals;
16          }
17
18  }
```

List 25 – ServiceVSteps class

Next, the definitions of the *ServiceV* predicates are introduced. They will be very familiar from the previous predicates. The *Assigned* predicate has the *Unassigned* state as precedent condition and a parameter (*svType*) that maps the *ComposedService* to which it was *Assigned*. If the *Service* has a *Payload* requirement, which means the *Vehicle Assigned* to the *ServiceV* has it. This predicate has an additional condition: it triggers the specified *Payload* predicate to *Assigned*.

```
1   ServiceV::Assigned
2   {
3           met_by(Unassigned);
4           any(object.vehicle.Assigned assigned)
5           assigned.ptype == ptype;
6
7           Payload payload;                    // If the Service has a payload requirement
8           payload.ptype == ptype;             //identifies the payload
9           payload.vehicle == object.vehicle;  //identifies the vehicle the payload belongs to
10          starts(payload.Assigned);
11
12  }
```

List 26 – Assigned predicate from ServiceV class

The *Executing* predicate is preceded by the *Assigned* one. The execution of the *Service* has, as said before, an action sequence, and so, in the *Executing* predicate, the first action is triggered. It can be seen that *Action1* starts along with the predicate and that *Action3* ends the *Service* execution. This triggers not only the *ServiceV* predicate *Unassigned*, but also ends the *Vehicle* assignment. In this predicate the *Goals* of the first action are also specified, and they derive from the *Goals* parameter or the identification of the *Vehicles Assigned* to the other *ComposedService* Services. Again, if the *ServiceV* has a *Payload* requirement, which means the *Vehicle Assigned* to the *Service* has it. This predicate has an additional condition: it triggers the *Payload* predicate to Unassigned.

```
1   ServiceV::Executing
2   {
3           met_by(Assigned assigned)
4           assigned.ptype == ptype;
5           starts(object.steps.Action1 a1);                    //starts the first action
6           a1.actgoal == goal1;
7
8           any(object.steps.Action3 a3);                       //last action
9
10          this.end == a3.end;
11          any(object.vehicle.Assigned a);
12          a.end == this.end;
13          meets(Unassigned);
14
15          Payload payload;                        // If the Service has a payload requirement
16          payload.ptype == ptype;
17          payload.vehicle == object.vehicle;
18          starts(payload.Assigned);
19  }
```

List 27 – Executing predicate from ServiceV class

The *ServiceVSteps* predicates follow the same logic of the predicates presented, but their conditions vary depending on its specification. The encoding presented here maps the *Gather* action shown in Figure 41. The contained by constraint ensures the *vehicle* is *Loitering* at the *Goal*, being this goal a *Location*, and the equals constraint ensures that the *Payload* indeed takes the *Sample* (being *Sample* a *PayloadActions*), and that it is taking a sample at the correct *Location*.

```
1   ServiceVSteps::Gather
2   {
3           contained_by(vehicle.navigator.Loiter loiter);
4           loiter.at == actgoal;
5
6           equals(vehicle.payload.actions.Sample sample);
7           sample.goal == actgoal;
8   }
```

List 28 – Possible action from ServiceVSteps class

Chapter 6

Results

In this chapter, the global results for the model application are discussed. The model was applied following the specifications presented in Chapter 5.

6.1 Simulation Examples

The model, as said in Chapter 4, was developed with the objective to define composed operations capable of providing specific system entities with the planning knowledge to establish them, and to provide support on system state capture for advanced missions. To get a better understanding on how the model is supposed to work on a real world multi-vehicle system, the model was applied to some operational scenarios examples.

6.1.1 Operational Scenario 1

A ground team (Team 1) wants to conduct a water temperature measure operation in specific locations. Team 1 fleet, a heterogeneous fleet, is composed by two UAV and three AUV's. Each one of these vehicles has a specific configuration. The team has some tight timing constraints that complicate their planning job:

- They need the water temperature information to be at their location before time T=85

And due to the AUV's velocity it is not possible to send them to all locations, get the samples, and have them at their location to transmit the data on time. Their solution is to use a fast relay vehicle, as the UAVs. But:

- The UAV1 is only available after time T=20 and UAV2 is unavailable to relay

6.1.2 Operational Scenario 2

This scenario is similar to the presented earlier, but in this case Team 1 wants to conduct both the water temperature measure operation and turbidity level measure operation in different locations. This time Team 1 has just the timing constraint to get all the data before time t=100. As in the scenario before the UAV is going to be used as a relay vehicle and the two AUV's are going to conduct the gather operations, being each assigned to one operation depending on its specification.

6.1.3 Operational Scenario 3

This last scenario assumes Team 1 wants again a water temperature measure and a bathymetry operation, but there is a need to get the bathymetry data in a different time then the temperature data:

- They need the bathymetry data to be at their location before time T=200
- They need the temperature data to be at their location before time T=200

Again, as in the scenarios before UAVs are going to be used as relay vehicles and AUVs as gather vehicles.

6.1.4 World Description

The world is assumed to be a Euclidean space grid, where the concept location was encoded in the *Location* class. The class has three attributes. The *name* is a symbolic name for the location and the *id* is an identification to distinguish between same name locations. The *x*, *y* and *z* attributes are coordinates. A vehicle can just move on one axis at each step.

```
1   class Location // A point on the planet's surface
2   {
3               string name;
4               int id;
5               int x;
6               int y;
7               int z;
8
9               Location(string _name, int _id, int _x, int _y, int _z)  {
10                      name = _name;
11                      id = _id;
12                      x = _x;
13                      y = _y;
14                      z = _z;
15              }
16  }
```

List 29 - class Location

Six agents populate the world, being each one of them defined following the proper model architecture (see section 5.2.1and 5.4.2.1):

- **CS1**

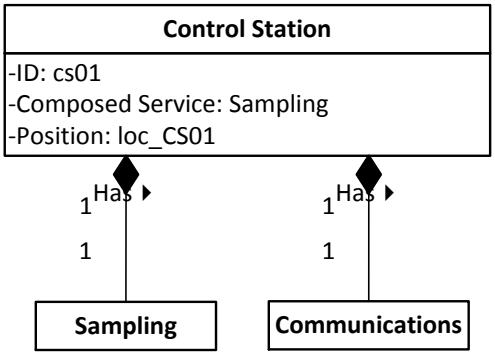

Figure 42 – CS1 model and attributes

- **UAV1**

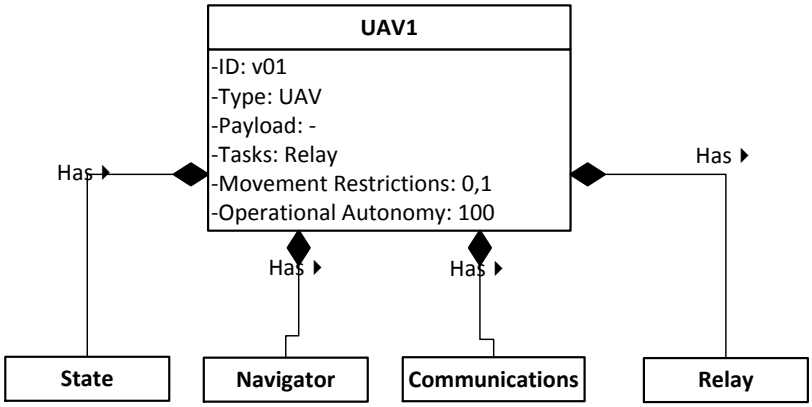

Figure 43 - UAV1 model and attributes

- **UAV2**

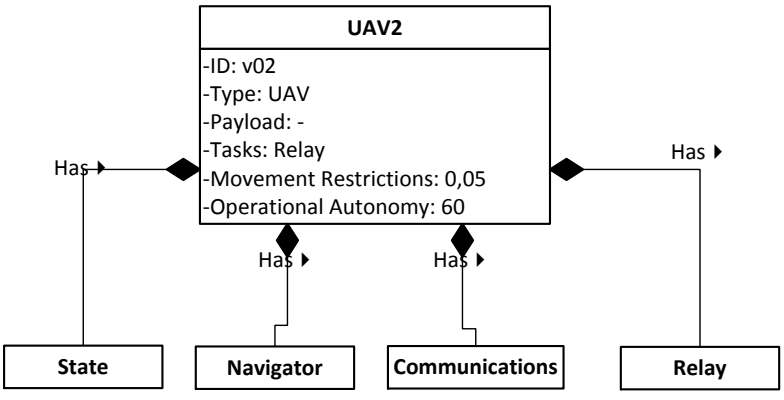

Figure 44 – UAV2 model and attributes

- **AUV1**

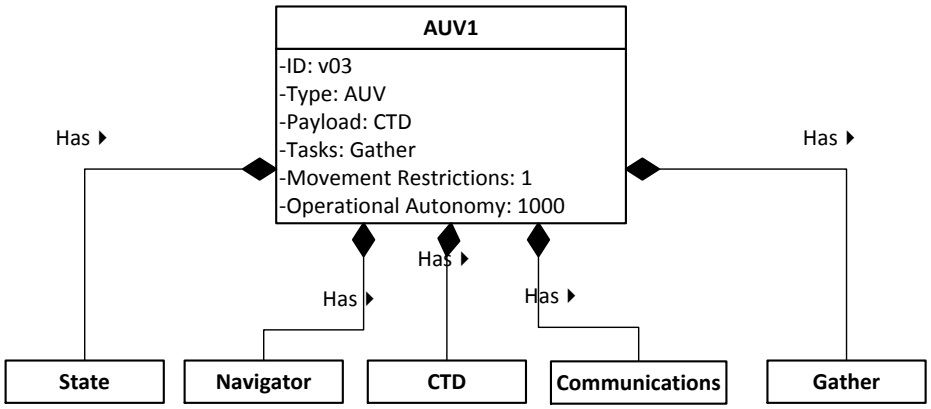

Figure 45 - AUV1 model and attributes

- **AUV2**

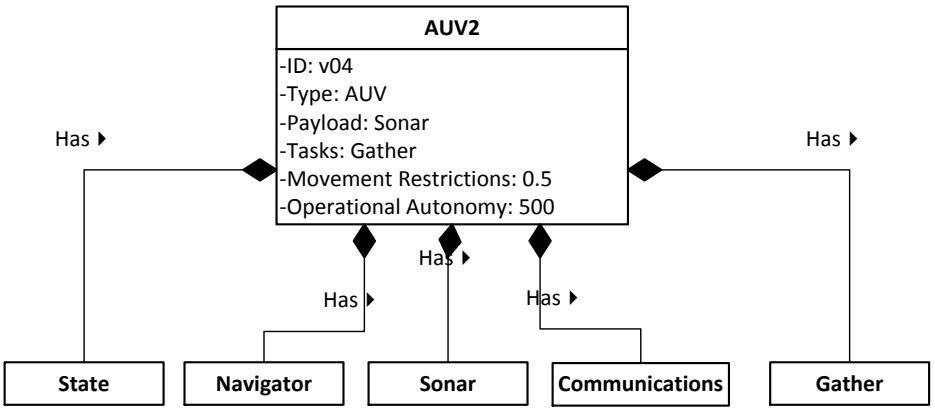

Figure 46 - AUV2 model and attributes

- **AUV3**

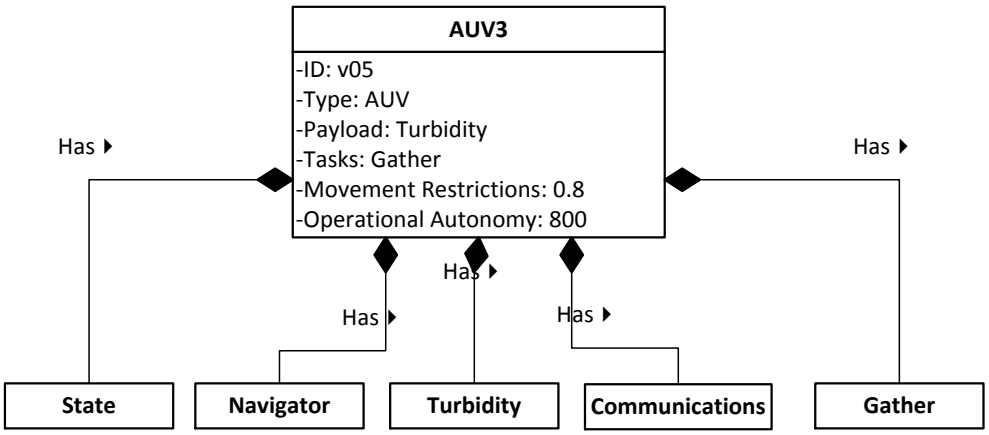

Figure 47 - AUV3 model and attributes

```
1   //Creating the world objects
2   ControlStation CS1 = new ControlStation ("cs01", cs_sampling,loc_CS1);        //CS1
3   Sampling CS1_sampling = new Sampling (CS1);
4   Vehicle UAV1 = new Vehicle("v01",UAV,none,0.1,s_relay);                       //UAV1
5   RelayV UAV1_relay = new RelayV(UAV1);
6   Vehicle UAV2 = new Vehicle("v02",UAV,none,0.1,s_relay);                       //UAV2
7   RelayV UAV2_relay = new RelayV(UAV2);
8   Vehicle AUV1 = new Vehicle("v03",AUV,1,s_gather);                             //AUV1
9   Payload AUV1_CTD = new Payload(AUV1,CTD);
10  GatherV AUV1_gather = new GatherV(AUV1);
11  Vehicle AUV2 = new Vehicle("v04",AUV,0.5,s_gather);                           //AUV2
12  Payload AUV2_Sonar = new Payload(AUV2,Sonar);
13  GatherV AUV2_gather = new GatherV(AUV2);
14  Vehicle AUV3 = new Vehicle("v05",AUV,0.8,s_gather);                           //AUV3
15  Payload AUV3_ Turbidity = new Payload(AUV2, Turbidity);
16  GatherV AUV3_gather = new GatherV(AUV3);
```

List 30 – Creating the world objects

And the world was configured to have several different locations:

- **CS1 location**: where the CS1 is (the same location as UAV1 and UAV2)
- **AUV's location**: where the AUV1, AUV2 and AUV3 are at initial time
- **Relay location**: where the vehicles should meet to relay the data from one to another
- **CTD Gather locations**: six locations to conduct the temperature measure
- **Sonar Gather locations**: three locations to conduct the bathymetry measure
- **Turbidity Gather locations**: three locations to conduct the turbidity measure

```
1   //Creating the world locations
1   Location loc_CS1 = new Location("CS1L", 0, 0, 0);
2   Location loc_relay = new Location("RL", 18, 19, 0);
3   Location loc_UAVS = new Location("UAVL", 16, 16, 0);
4   Location loc_CTDgather1 = new Location("CTD", 1, 18, 18, -5);
5   Location loc_CTDgather2 = new Location ("CTD", 2, 19, 18, -5);
6   Location loc_CTDgather3 = new Location ("CTD", 3, 20, 18, -5);
7   Location loc_CTDgather4 = new Location ("CTD", 4, 20, 19, -5);
8   Location loc_CTDgather5 = new Location ("CTD", 5, 19, 19, -5);
9   Location loc_CTDgather6 = new Location ("CTD", 6, 18, 19, -5);
10  Location loc_Sonargather1 = new Location("Sonar", 1, 18, 20, -5);
11  Location loc_Sonargather2 = new Location ("Sonar", 2, 19, 20, -5);
12  Location loc_Sonargather3 = new Location ("Sonar", 3, 20, 20, -5);
13  Location loc_Sonargather3 = new Location ("Sonar", 4, 19, 18, -5);
14  Location loc_Turbgather3 = new Location ("Turbidity", 1, 21, 18, -5);
15  Location loc_Turbgather3 = new Location ("Turbidity", 2, 21, 19, -5);
16  Location loc_Turbgather3 = new Location ("Turbidity", 3, 21, 19, -5);
```

List 31 – Creating the world locations

6.1.5 Initial State

6.1.5.1 Operational Scenario 1

The initial system state is created by placing tokens on the objects created. It is assumed that at time zero:

- Each *Vehicle* is at the positions referred before
- Each *Vehicle* state is *Unassigned* (see List 7), although *UAV1* has to be *Unassigned* until time T=20
- *Sampling ComposedService* is *Available* (see List 16)

```
1   //ComposedService available at time 0
2   fact(CS1_SAMPLING1.Available sampling_available);
3   eq(CS1_SAMPLING1_available.start,0);
4   //UAV1 can only be assigned to a Service after time 20 and the other vehicles can be  assigned after time 0
5   fact(UAV1.Unassigned UAV1_unassigned);
6   eq(UAV1_unassigned.start,0);
7   lt(20, UAV1_unassigned.end);
8   fact(UAV2.Unassigned UAV2_unassigned);
9   eq(UAV2_unassigned.start,0);
10  fact(AUV1.Unassigned AUV1_unassigned);
11  eq(AUV1_unassigned.start,0);
12  fact(AUV2.Unassigned AUV2_unassigned);
13  eq(AUV2_unassigned.start,0)
14  fact(AUV3.Unassigned AUV3_unassigned);
15  eq(AUV3_unassigned.start,0);
16  //Positioning the vehicles at initial position at time 0
17  fact(UAV1.navigator.Loiter UAV1IniPosition);      //UAV1
18  eq(UAV1IniPosition.start, 0);
19  eq(UAV1IniPosition.location, loc_delivery);
20  fact(UAV2.navigator.Loiter UAV2IniPosition);      //UAV1
21  eq(UAV2IniPosition.start, 0);
22  eq(UAV2IniPosition.location, loc_delivery);
23  fact(AUV1.navigator.Loiter AUV1IniPosition) ;     //AUV1
24  eq(AUV1IniPosition.start, 0);
25  eq(AUV1IniPosition.location, loc_UAVS);
26  fact(AUV2.navigator.Loiter AUV2IniPosition);      //AUV2
27  eq(AUV2IniPosition.start, 0);
28  eq(AUV2IniPosition.location, loc_UAVS);
29  fact(AUV3.navigator.Loiter AUV3IniPosition);      //AUV3
30  eq(AUV3IniPosition.start, 0);
31  eq(AUV3IniPosition.location, loc_UAVS);
```

List 32 – Defining the initial state, operational scenario 1

And the goal is defined as having the Sampling *ComposedService Completed* between time 0 and 70. The *ComposedService* parameters are also defined.

```
1   //Defining the goals
2   goal(CS1_SAMPLING1.Completed goal1);
3   goal1.ptype == CTD;                              //Defining the gather payload
4   lt(0,goal1.start);
5   lt(goal1.start,85);
```

List 33 – Defining the goals – Operational Scenario 1

6.1.5.2 Operational Scenario 2

The same initial state as the scenario before but with a different composed service, it has two gather services and one relay service, and the UAV1 has no restrictions. The goals are:

```
1  goal(CS1_DOUBLESAMPLING.Completed goal3);
2  goal3.gather1_stype == Turbidity;
3  goal3.gather2_stype == Sonar;
4  lt(0,goal3.start);
5  lt(goal3.start,100);
```

List 34 – Defining the goals – Operational Scenario 2

6.1.5.3 Operational Scenario 3

The same initial state as the scenario before but with two composed services equal to the composed service at scenario 1. The goals ares:

```
1  goal(CS1_SAMPLING1.Completed goal1);
2  goal1.ptype == CTD;                    //Defining the gather payload
3  lt(0,goal1.start);
4  lt(goal1.start,200);
5
6  goal(CS1_SAMPLING2.Completed goal2);
7  goal1.ptype == Turbidity;              //Defining the gather payload
8  lt(0,goal2.start);
9  lt(goal2.start,200);
```

List 35 – Defining the goals – Operational Scenario 3

6.2 Solver Results

The scenarios were solved using the standard EUROPA solver. It assumes a chronological-backtracking, heuristically guided, refinement search. The solver window shoes the number of step decisions made (Step Count) as well as the deliberative time to output the plan (Run time). The tree consecutive window inside the solver window show the deliberative time at each step (Time (secs) per Step), the decisions to be made to achieve the goal (Open Decision Count) and the decisions made to output the plan (Decisions in Plan). It is important to highlight that when a plan is successfully generated the Open Decision Count gets to zero.

From the comparison of the solver decisions (Figure 48, Figure 49 and Figure 50) it is possible to observe that the solver takes more time to plan when the decisions to be made to output a plan are bigger. At these three scenarios it is possible to see that the solver gets to the solution without backtracking any solution (the decisions in plan always grow with the time).

Figure 50 shows that in scenario 3 the solver has to backtrack some decisions and consequently increases the deliberative time for those decisions. It is possible to see that the

steps count is higher than any other case and the run time much higher, more than 30 times higher than the slowest scenario among the others (Figure 49).

Figure 48 - Solver for Operational Scenario 1

Figure 49 - Solver for Operational Scenario 2

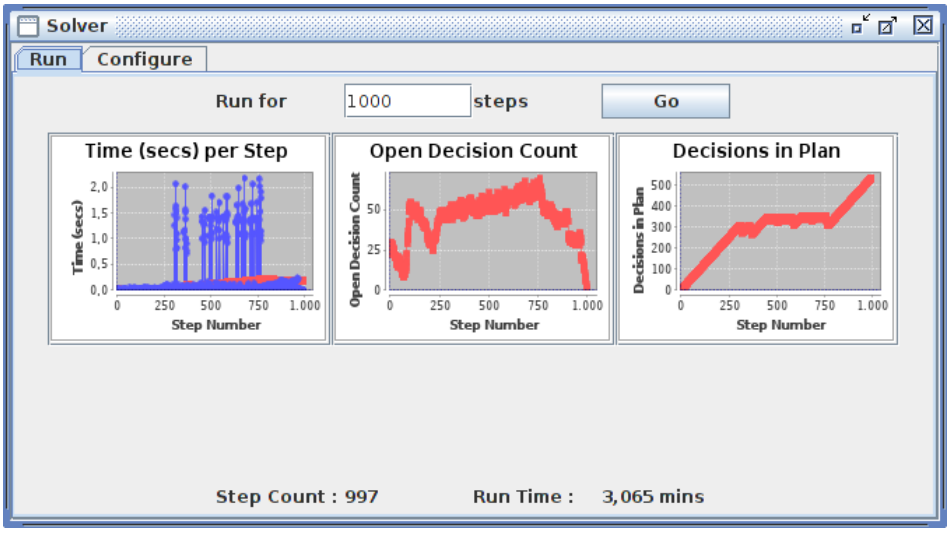

Figure 50 - Solver for Operational Scenario 3

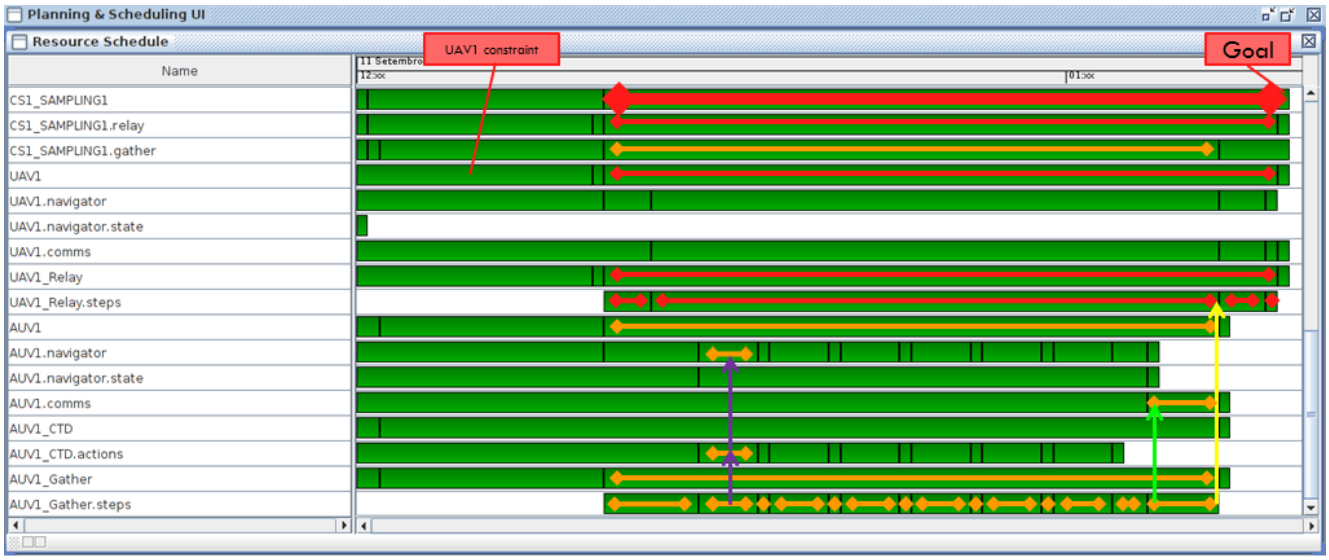

Figure 51 - EUROPA UI showing a domain solution for operational scenario 1

6.3 Scheduler Results

6.3.1 Operational Scenario 1

In Figure 51, the resulting schedule for operational scenario 1 is shown, as a Gantt chart. The green rectangles map the actions over each timeline. If the mouse is over any of them, it is possible to see the details displayed in the Details window. For example, the details shown prove that:

- The *Sampling* ComposedService is completed at time T=[79,80] (from Figure 52 it is possible to see that the relay vehicle ends execution at T=[78,79] and so the state starts after it)
- *UAV1* is *Executing* the *Relay* service just at time T=21, see Figure 52 (parameter *svtype* maps the assigned *Service*, see List 7 and 22)

Observing the chart it is also understandable that:

- The *Relay Service* execution duration equals the *UAV1 Relay Service* execution and the *UAV1 Executing* state (filled red rectangles), and all the *Vehicle Relay Service Steps* are contained by it
- The *Gather Service* execution duration equals the *AUV1 Gather Service* execution and the *AUV1 Executing* state (filled orange rectangles), and all the *Vehicle Gather Service Steps* are contained by it
- From the two available AUVs, the AUV1 is chosen because is the one that meets the payload requirement (see List 35)

- The *Sampling* execution state equals the longest *Service*, in this case the *Relay Service* (filled red rectangles)

- The *Gather Step* that correspond to the *Gather* predicate trigger the payload *CTD* to execute the defined action and it is contained by the navigator at the gather location, the details are shown in Figure 54 (purple arrows)

- The *Gather Step* that correspond to the *SendToRelay* predicate trigger the Communications timeline to execute the Send action (light green arrow) and after completion, it triggers the Receive predicate on the AUV1 Communications timeline to end, details are shown in Figure 53 (yellow arrow).

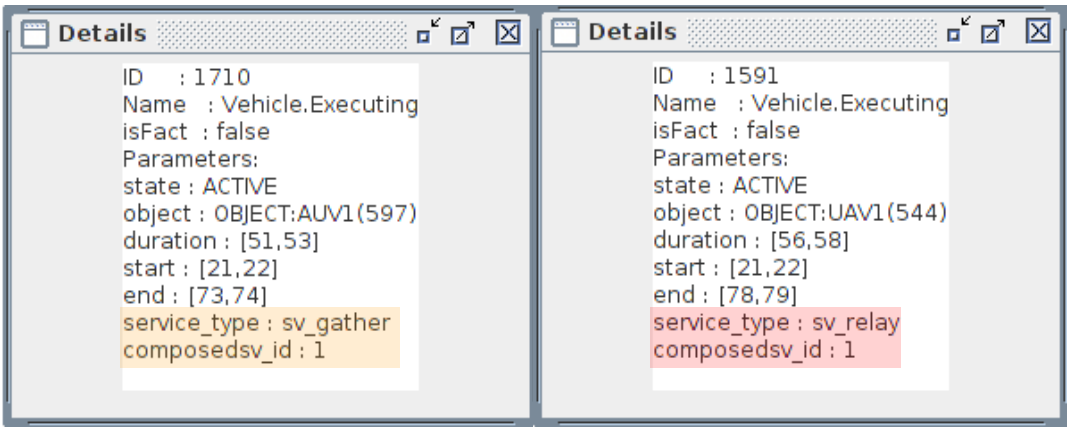

Figure 52 - Details of the vehicles Executing state

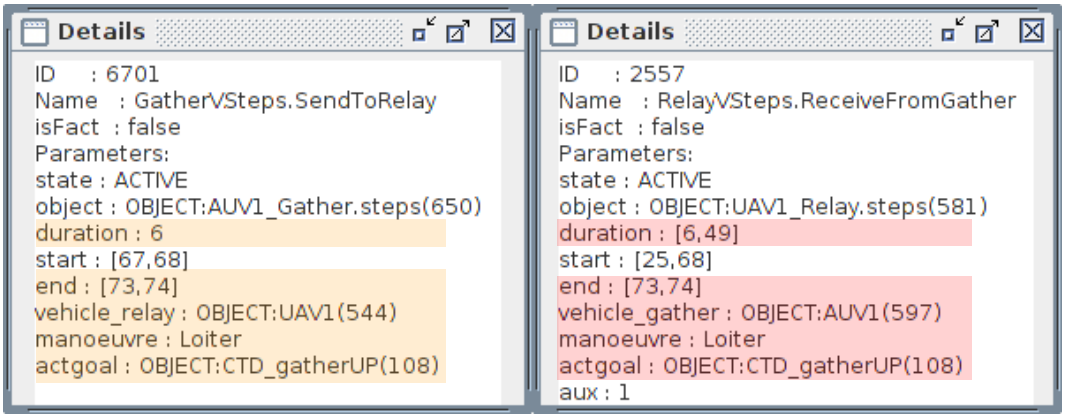

Figure 53 – Details of the coordinated services steps

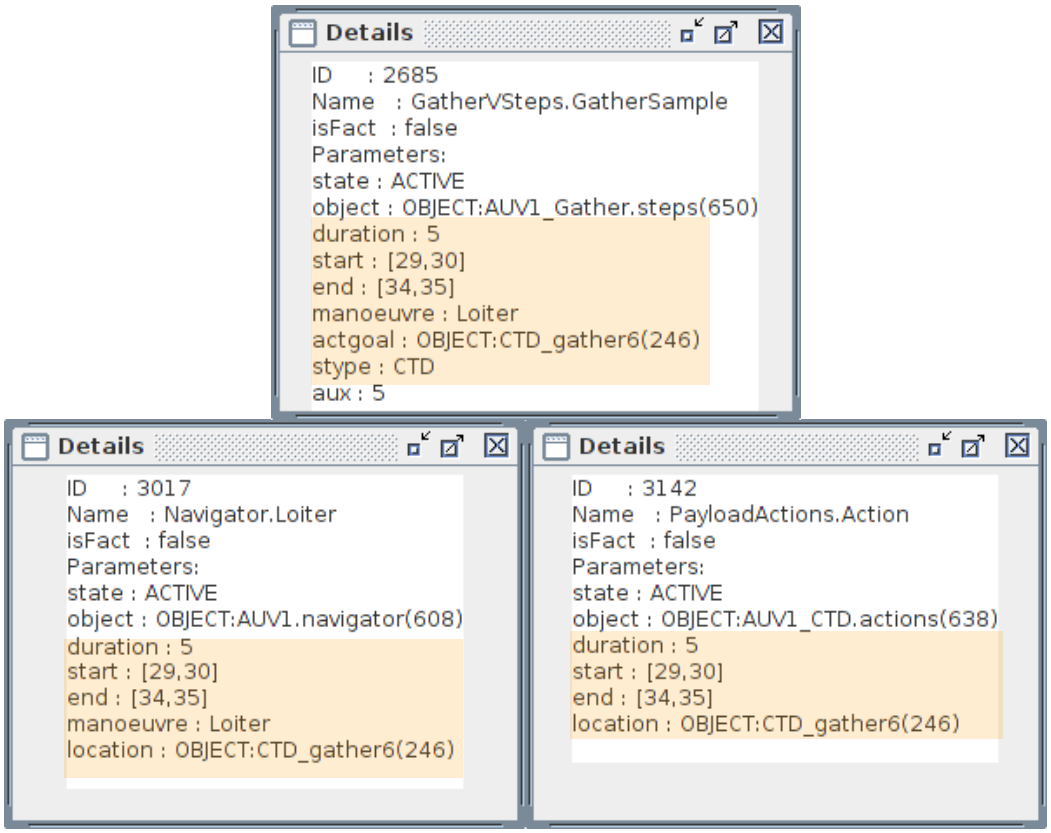

Figure 54 - Details of the Gather Step GatherSample

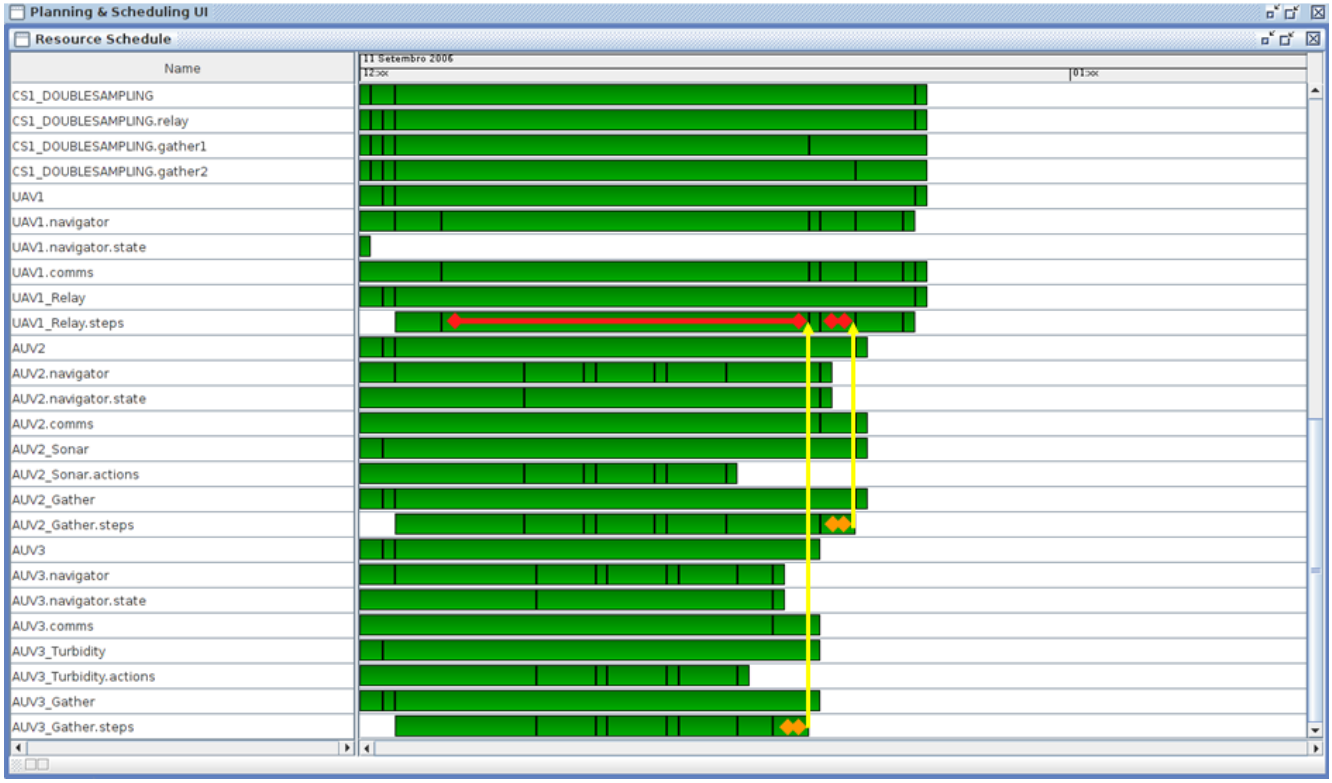

Figure 55 - EUROPA UI showing a domain solution for operational scenario 2

6.3.2 Operational Scenario 2

In Figure 55, the resulting schedule for operational scenario 2 is shown. From these results it is possible to prove that:

- The *Relay Vehicle* successfully "receives" information from the two vehicles, even though there is no information on each to go first (yellow arrows)

6.3.3 Operational Scenario 3

In Figure 56, the resulting schedule for operational scenario 3 is shown. From these results it is possible to understand that:

- The solver uses the same *Vehicle* (UAV1) to execute the *Relay* service in both the composed services

Consequently this choice leads to a worst solution than if the *Relay* services were assign to different vehicles (UAV1 and UAV2). The red goal would have been achieve in almost half the time, as blue goal shows.

This solution is only possible because the goals timing constraints (T=200) are loose enough to fit both composed services *Relay* service in the same vehicle. It was also tested this scenario with more tight timing constraints and it was concluded that the solver always tries to assign both composed services Relay service to the same vehicle. This happens because the solver does not know the time to complete the *Gather* services before choosing the *Relay* vehicle and so it assumes there is time to fit both services in the same vehicle. When the solver gets to the point where both *Gather* services have their duration known, the decision tree is already too big to backtrack to the choosing *Relay* vehicle choice.

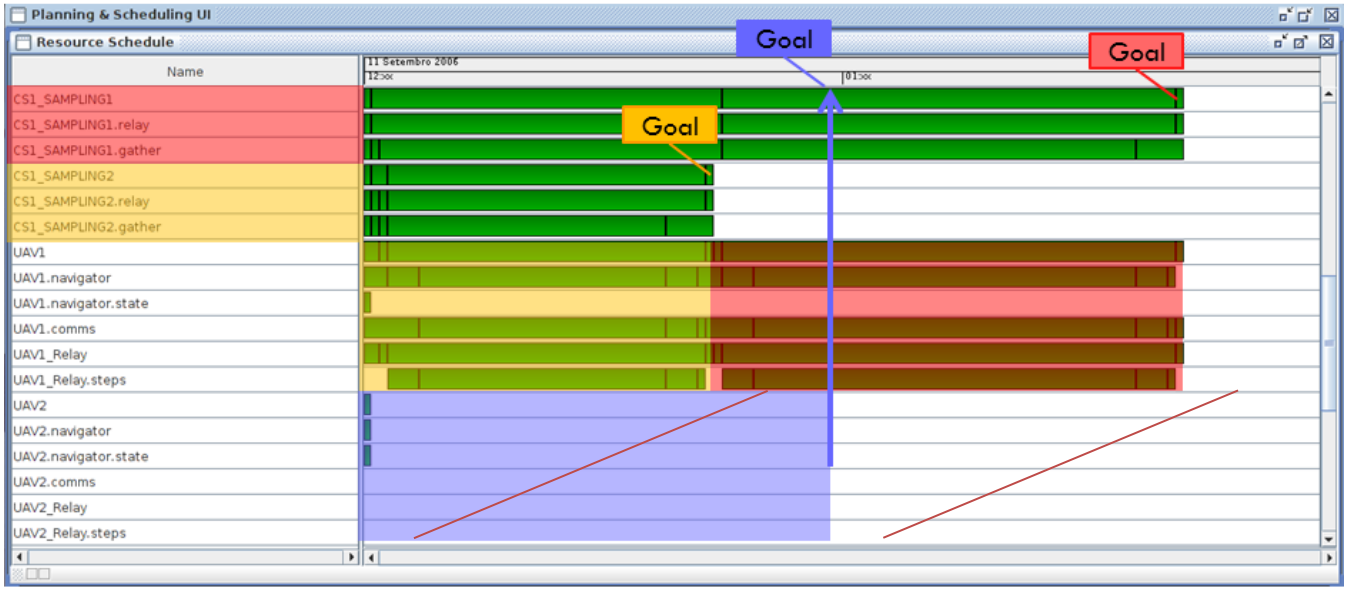

Figure 56 - EUROPA UI showing a domain solution for operational scenario 3

At Figure 57 it is possible to see this case, where after 36 minutes the solver is still trying to find a solution, being with 5000 choices made. If the run continued the solver would eventually explode because of memory fault.

There are some possible solutions to fix this case, being one of them tuning the solver to the described problem.

Figure 57 - Solver for Operational Scenario 3 with no solution

Chapter 7

Conclusions and Future Work

7.1 Summary

This thesis started with the objective to develop a new model for standardization of cooperative operations plans along with providing support on defining the system state for complex mission. In order to accomplish this proposes, the following steps were made:

- Initial study of the background of multi-vehicles network, in order to understand the main motivations on deploying multi-vehicles systems
- Analysis of vehicle cooperative missions to understand how these are currently deployed
- State of the Art review, to comprehend and learn from what have been developed on past researches on multi-vehicle systems and planning paradigms
- Understand the crucial requirements when capturing a system state
- Build initial models for the system entities based on those requirements
- Think of how cooperative configurations could be composed in a model
- Build an initial cooperative concept model
- Developing each model step by step, involving multiple iterations, until reaching a desired complete, combined model
- After having the initial concept model, encode the model, also involving multiple iterations

- Finally, some simulations were made on the model to prove its consistency and the new introduced concepts

7.2 Model Evaluation

The model is going to be evaluated and analyzed along with the multi-vehicle issues presented at section 3.1.5.

- **System model:**
  - **Is there a defined control hierarchy? How is it defined?**

  → There is a control hierarchy defined thought the Composed Service model where the goals are sent top down the model and the low nodes results/achievements are sent to the upper nodes.

  - **Does cooperation exists? Is it crucial in achieving system goals?**

  → Cooperation is also defined thought the Composed Service model with the specification of where each Service interacts with another. This creates a Service dependency that is essential to the system to achieve the goals.

  - **Does cooperation between agents forms an agent composed model? What benefit comes from achieving the composed model? Who knows when a model is composed?**

  → As described in the last two points, agents' cooperation is defined through the Composed Service model. When a Composed Service model is achieved, an additional feature becomes available in the system, being its achievement managed by the requesting agent. This agent has the Composed Service specification knowledge.

  - **What and where are the communicating structures?**

  → The communicating structures belong to any agent that implements a communicating system, as vehicles and control stations.

  - **Problem-solving Strategy: How are system goals distributed and broken down to sub goals? Are all goals defined in the planning phase?**

  → As said before the goals are distributed top down along the Composed Service model, being the goals distribution defined in the Composed Service specification. Some goals are defined in the planning phase while others arise while executing.

- **System domain and environment assumptions:**
  - **How is the system domain characterized?**

  → The system domain is composed by all the agents and their components, being physical entities, computational agents or abstract concepts

  - **What a priori knowledge do agents have about other agents?**

  → Service requesting agents need to know the other agents configuration

  - **How is problem solving knowledge shared among agents?**

  → The structure of knowledge differs between agents and algorithms or procedures may be entirely different.

  - **Is the system domain dynamic? What features may vary along time?**

  → The system domain may vary, although the problem solving always refer to the initial domain state used

- **Fault tolerance:**
  - **What possible failures are assumed to be possible?**

  → No possible failures are assumed at the moment

  - **How does the system deals with faults?**

  → No fault tolerance system has been developed.

  - **Is predictive timing information available?**

  → Timings are known or computed at various levels of accuracy at the start of problem solving.

  - **Are there tasks deadlines?**

  → Task deadlines are introduced at the planning phase.

- **Communication importance:**
  - **How are the communication paths specified?**

  → Agents know who they may communicate with beforehand, but the order and contents of that communication are not pre-specified.

- **How are messages addressed?**

→ A full range of methods is available to use depending on the situation (desirable).

- **What information do agents use to determine the when, whom, and what of communication?**

→ A combination of long-term, fixed knowledge and local knowledge developed during problem solving.

7.3 Achieved Goals

The proposed initial objective was successfully achieved. A model was developed that could provide a standardization for planning composed cooperative operations, providing system with the knowledge of how to achieve known composed operations, and improve the system state capture.

The main contribution this work lays down on multi-vehicle systems is:

- Standardized composed cooperative operations models, which specify how these are achieved and how its execution flows, provide systems with the planning capacities to successfully output complex plans

Although the model proved to achieve good results on the scenarios described, there are some weaknesses on the implementation provided, as already seen on the previous section:

- The scenarios are assumed to be unchanged over planning and execution time, what means no new objects can be created after the initial state is defined

- It is not possible to implement a service to map complex cooperative networks, as for example flight formation controllers. Although, it would be possible to define a service that would request the execution of a flight formation controller with specific parameters and time restrictions

- When the model objects increase and consequently the decisions to be made to output a plan, the computational effort along with the time consumed also increase

- At the moment, there is no fault-tolerance, although the outputted plans have temporal windows where actions have to happen, which may solve some minor temporal issues as moving and actions executing delays

7.4 Future Work

The multi-vehicle system is a growing research area with much improvements and developments to be done. This work presented a new approach on planning complex operations with some level of cooperation and even though the model provided a good answer for the simulated operational scenarios a lot of improvements and further developments have to be done, among them:

1. Continue to grow the model to fit more complex operations and develop a connection that could bring complex controllers, as flight formation, persistence surveillance, obstacle avoidance, into the planning loop

2. Along with the above point, it would be important to implement the model such that it could deal with dynamic environments, adding a re-planning feature

3. Deal more detailed with the communications issues that were not considered during this work, as already approached in the last section. How messages are addressed between the communication structures? How are the communication paths specified? How to deal with communication losses? Among other issues that certainly arise when operating real world scenarios

4. Shape the model to be implemented with the current onboard deliberative planning tool chain available at LSTS: the TREX teleo-reactor executive and the EUROPA planner [40]

5. Field-tested the model, from a simple to a more complex version to prove its effectiveness when planning real world scenarios

References

[1] P. D. E. Investiga, T. E. M. Ve, and R. Aut, “II Série Cadernos do IDN,” vol. 2008, pp. 9–24, 2009.

[2] F. L. Pereira, P. F. Souto, and L. Madureira, “DISTRIBUTED SENSOR AND VEHICLE NETWORKED SYSTEMS FOR ENVIRONMENTAL APPLICATIONS,” no. May 2003, pp. 1–6, 2010.

[3] J. Pinto, P. S. Dias, R. Gonçalves, E. Marques, G. Gonçalves, J. B. Sousa, and F. L. Pereira, “Neptus – A Framework to Support a Mission Life Cycle,” in *7th IFAC Conference on Manoeuvring and Control of Marine Craft*, 2006.

[4] “Laboratório de Sistemas e Tecnologia Subaquática.” [Online]. Available: http://whale.fe.up.pt/main/tech.

[5] M. E. Angelopoulou, R. Martins, J. B. de Sousa, S. Chatzichristofis, L. Doitsidis, M. Kothari, M. G. Lagoudakis, L. Panagiotopoulou, M. Petrou, V. S. Prabhu, P. B. Sujit, and C. Tsiotsios, “NOPTILUS : System Specifications,” 2012.

[6] P. S. Dias, R. M. F. Gomes, and F. L. Pereira, “Mission Planning and Specification in the Neptus Framework,” no. May, pp. 3220–3225, 2006.

[7] M. Correia, P. Dias, S. Fraga, R. Gomes, R. Goncalves, L. Madureira, F. L. Pereira, R. Picas, J. Pinto, A. Santos, A. Sousa, and J. B. de Sousa, “OPERATIONS AND CONTROL OF UNMANNED UNDERWATER VEHICLES,” 2005.

[8] R. Martins, P. S. Dias, E. R. B. Marques, J. Pinto, J. B. Sousa, and F. L. Pereira, “IMC: A communication protocol for networked vehicles and sensors,” *Oceans 2009-Europe*, pp. 1–6, May 2009.

[9] S. A. Bortoffl, S. Lane, and E. C. T. Hartford, “Path Planning for UAVs,” no. June, 2000.

[10] Y. E. H. Bang, “Cooperative Control of Multiple Unmanned Aerial Vehicles Using the Potential Field Theory.” 2006.

[11] Y. E. H. Bang, “Cooperative Task AssignmentPath Planning of Multiple Unmanned Aerial Vehicles Using Genetic Algorithm.” 2009.

[12] Z. Hu, M. Zhao, and M. Yao, “Cooperative Attack Path Planning for Unmanned Air Vehicles Swarm Based on Grid Model and Bi-level Programming Grid-based Space Division Coordinated Attack Model by Bi-level Programming,” vol. 4, no. 2009, pp. 671–679, 2011.

[13] T. Schouwenaars, B. D. Moor, E. Feron, and J. How, “MIXED INTEGER PROGRAMMING FOR MULTI-VEHICLE PATH PLANNING.”

[14] S. Leary, M. Deittert, and J. Bookless, “Constrained UAV Mission Planning : A Comparison of Approaches,” pp. 2002–2009, 2011.

[15] A. B. M. I. L. Pollini, “Cooperative Task Assignment Using Dynamic Ranking,” pp. 5712–5717, 2008.

[16] W. A. I. R. F. Base, “UAV TASK ASSIGNMENT WITH MIXED-INTEGER LINEAR PROGRAMMING,” 2004.

[17] Y. Jin, A. A. Minai, and M. M. Polycarpou, “Cooperative Real-Time Search and Task Allocation in UAV Teams.”

[18] J. Tao and Y. Tian, “Cooperative Task Allocation for Unmanned Combat Aerial Vehicles Using Improved Ant Colony Algorithm,” pp. 1220–1225, 2008.

[19] C. Schumacher, M. Pachter, and W. A. I. R. F. Base, “UAV TASK ASSIGNMENT WITH TIMING CONSTRAINTS,” 2003.

[20] Z. Papp, C. Brown, and C. Bartels, “World modeling for cooperative intelligent vehicles,” *Intelligent Vehicles Symposium*, pp. 1050–1055, 2008.

[21] H. Kawakami and T. Namerikawa, “Cooperative target-capturing strategy for multi-vehicle systems with dynamic network topology,” *2009 American Control Conference*, pp. 635–640, 2009.

[22] M. Zennaro, J. Ko, R. Sengupta, and S. Tripakis, “A service network architecture for a multi-vehicle search mission,” *Proceedings of the 40th IEEE Conference on Decision and Control (Cat. No.01CH37228)*, vol. 2, pp. 1503–1508, 2001.

[23] M. Dorigo, V. Maniezzo, and a Colorni, “Ant system: optimization by a colony of cooperating agents.,” *IEEE transactions on systems, man, and cybernetics. Part B, Cybernetics : a publication of the IEEE Systems, Man, and Cybernetics Society*, vol. 26, no. 1, pp. 29–41, Jan. 1996.

[24] J. B. Ioana Gheţa, Michael Heizmann, Andrey Belkin, “World Modeling for Autonomous Systems,” in *KI 2010: Advances in Artificial Intelligence*, 2010, pp. 176–183.

[25] K. S. Decker, E. H. Durfee, and V. R. Lesser, “Evaluating Research in Cooperative Distributed Problem Solving,” no. August, 1988.

[26] D. E. Smith, “Planning as an Iterative Process †.”

[27] R. Bart, M. A. Salido, and F. Rossi, *New Trends in Constraint Satisfaction , Planning , and Scheduling : A Survey*, vol. 00. 2004, pp. 1–24.

[28] T. Dean, “Automated planning,” *ACM Computing Surveys*, vol. 28, no. 1, pp. 85–87, Mar. 1996.

[29] “EUROPA.” [Online]. Available: https://code.google.com/p/europa-pso.

[30] R. Dechter, I. Meiri, and J. Pearl, “Temporal constraint networks,” *Artificial intelligence*, 1991.

[31] J. Frank and J. Ari, “Constraint-based Attribute and Interval Planning,” pp. 1–32, 2002.

[32] N. Museettola, “HSTS : Integrating Planning and Scheduling,” 1993.

[33] J. Barreiro, M. Boyce, M. Do, J. Frank, M. Iatauro, T. Kichkaylo, P. Morris, J. Ong, E. Remolina, T. Smith, and D. Smith, “EUROPA : A Platform for AI Planning , Scheduling , Constraint Programming , and Optimization,” 2004.

[34] N. Muscettola, P. P. Nayak, B. Pell, and B. C. Williams, “Artificial Intelligence Remote Agent : to boldly go where no AI system has gone before *,” vol. 103, no. 98, 1998.

[35] M. Ai-chang, J. Bresina, L. Charest, J. Hsu, K. J. Ari, B. Kanefsky, P. Maldague, P. Morris, K. Rajan, and J. Yglesias, “MAPGEN Planner : Mixed-initiative activity planning for the Mars Exploration Rover mission,” 2004.

[36] D. Tran, S. Chien, R. Sherwood, R. Castano, B. Cichy, A. Davies, and G. Rabideau, “DEMO : The Autonomous Sciencecraft Experiment Onboard the EO-1 Spacecraft,” pp. 1–2, 2005.

[37] C. Nicola Muscettola, Ben Smith and and D. Y. Fry, Steve Chien, Kanna Rajan, Gregg Rabideau, “On-board planning for new millenniumdeep space one autonomy,” in *IEEE Aerospace Conference*, 1997.

[38] K. Rajan, “Towards Deliberative Control in Marine Robotics.”

[39] J. F. Allen, “Maintaining knowledge about temporal intervals,” *Communications of the ACM*, vol. 26, no. 11, pp. 832–843, Nov. 1983.

[40] P. S. Dias, R. Martins, and E. Marques, “The LSTS Toolchain for Networked Vehicle Systems *.”